

SINGLE-PASS MULTI-VIEW RENDERING

Thomas Hübner, Yanci Zhang and Renato Pajarola *Visualization and Multimedia Lab,
Department of Informatics, University of Zurich, Binzmühlestrasse 14, CH-8050 Zürich*
huebner@ifi.uzh.ch , zhang@ifi.uzh.ch , pajarola@acm.org

ABSTRACT

The fundamental drawback of current stereo and multi-view display techniques is the necessity to perform multi-pass rendering (one pass for each separate view) and subsequent image composition-and-masking for generating multi-stereo views. Thus the rendering time increases in general linearly with the number of rendered views. An increase in frame-rate can be achieved by sub-resolution rendering and extrapolation before display, at the expense of degraded image quality.

To overcome these disadvantages, we introduce a new method for rendering volumes, points and triangles meshes on multi-view auto-stereoscopic displays. Our method exploits the programmability of modern graphic processing units (GPUs) for rendering multi-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 advantages of our single-pass multi-view rendering algorithms compared to the standard multi-pass rendering approaches.

KEYWORDS

Multi-View Rendering, Volume Visualization, Point Splatting, Triangle Meshes, Three-Dimensional Displays

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 little attention in the field. In multi-view display systems, rendering performance is generally an 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 [Ebert et al 1996, Miniel et al 2004]. Correspondingly, an N -view display requires a rendering time of up to $N \cdot t$ [Portoni et al 2000]. This overhead prohibits effective multi-view rendering of large data sets.

Specific rendering algorithms for multi-view autostereoscopic displays have not been proposed so far. Consequently, as done to date (e.g. SGI *Interactive Stereo Library* [Kakimoto 2005]), N views need to be rendered individually, and then combined and masked for a multi-view auto-stereoscopic display.

We address the problem of efficiently generating N views simultaneously in the context of volume rendering, point splatting and triangle meshes by presenting a solution that exploits the programmability of modern GPUs to calculate the multiple views directly on a per-fragment basis. Since our approach is a single-pass rendering algorithm, it requires approximately $K \cdot t$ rendering time independent of the display resolution and the number N of views. The constant K indicates the number of different views per-pixel, i.e. $K=3$ if the display system supports RGB sub-pixel views. This paper extends our previous multi-view point and volume rendering algorithms [Hübner et al 2006, Hübner and Pajarola 2007] to include triangle based rendering and explains the basic ideas in a unified context from a general multi-view rendering point of view. Furthermore, we present additional details such as detailed vertex and fragment shader programs as well as experimental results.

The remainder of the paper is organized as follows. The theory for multi-view auto-stereoscopic displays is provided in Section 2. This is followed by the description of our multi-view rendering algorithms in Section 3. Section 4 reports experimental results for different test data sets and multi-view rendering algorithms. Finally, our paper is concluded in Section 5.

2. MULTI-VIEW RENDERING

2.1 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 simultaneous observers and view positions, in contrast to active systems which track a single 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 1, the infinite number of views an 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.

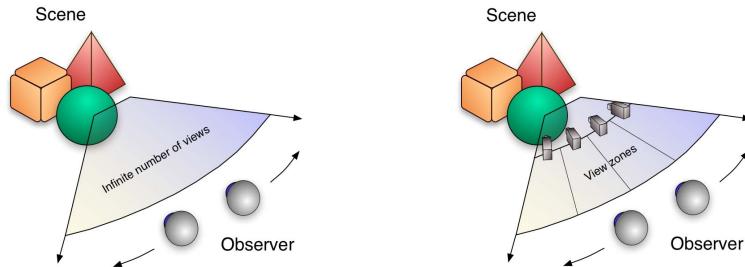


Figure 1. Auto-stereoscopic display principle: Discretization of an infinite view space into finite number of view zones

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 [McAllister et al 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 2, 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.

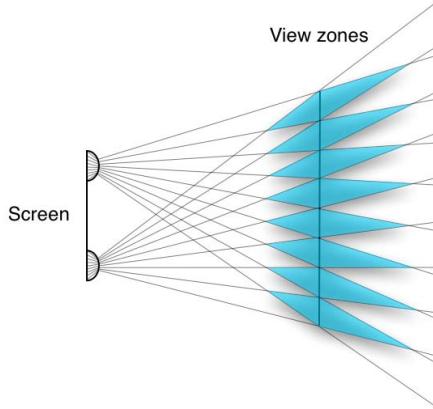


Figure 2. Viewing zones converge at a given observer distance

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 $x \bmod N$. However, using wavelength-selective filters, the reduction in horizontal resolution is limited to

$$X_{Res} = \frac{m}{N} \cdot 3$$

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.

We explain our multi-view rendering algorithms in the context of auto-stereoscopic displays featuring such wavelength-selective filter arrays. Though the presented methods can easily be adopted to 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.

2.2 View Geometry

In Figure 3a 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 3b 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.

As shown in Figure 3b, 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 3b, 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 3b to the convergence distance indicated by the display device.

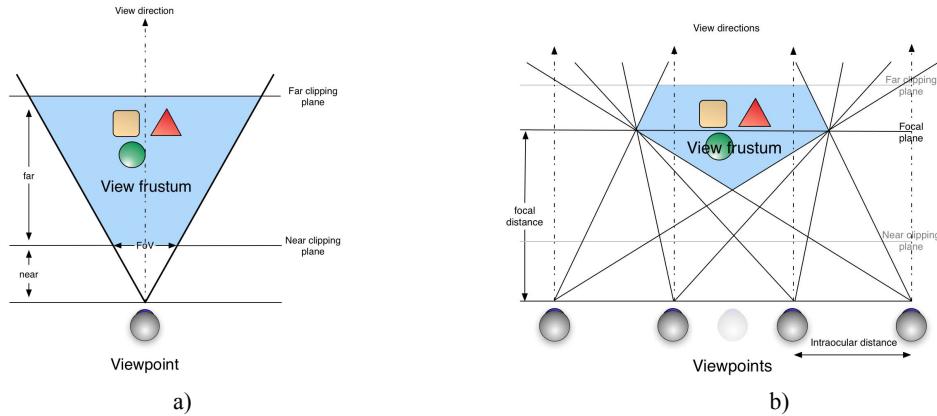


Figure 3. a) Single-view frustum with near and far clipping planes and b) Visible multi-view frustum as intersection of individual off-axis asymmetric sheared view frusta

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. 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 of a geometric primitive -- i.e. point on a planar surface -- visible in a view v_i depends on the surface normal n and the primitive's position as illustrated in Figure 5a and 5b. In the simple case, where the geometric primitive is parallel to the camera plane (e.g. used in direct volume rendering based on planar proxy geometries) we only have to consider the offset off_i parallel to the camera plane as illustrated in Figure 4.

Values contributing to a pixel originate from different positions on the geometric primitive. Furthermore, the projections of the same pixel in multiple views may not necessarily all be located on the (same) primitive (Figure 5a and 5b, point q_i).

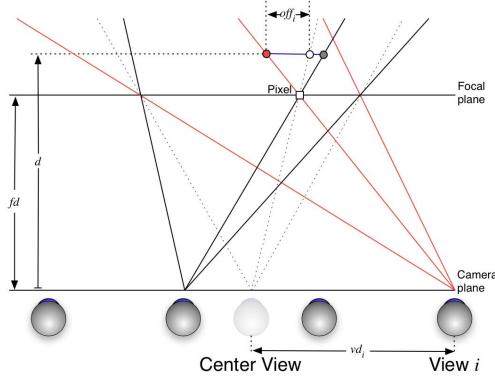


Figure 4. Multi-View volume parallax geometry

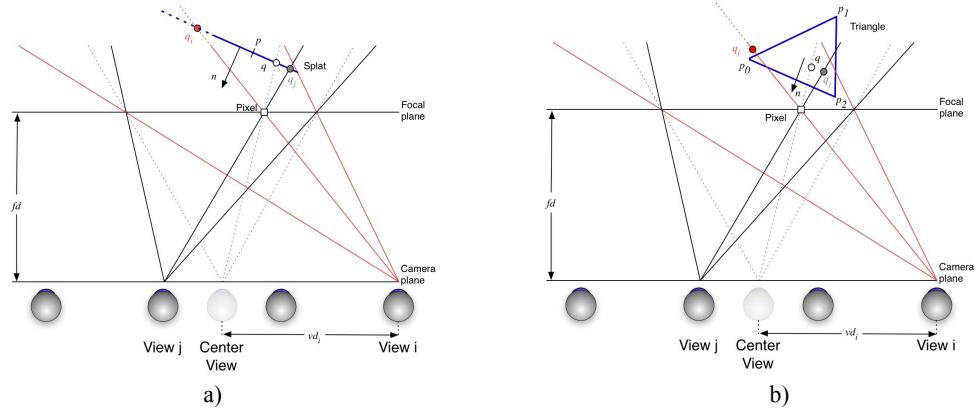


Figure 5. a) Multi-View point splatting parallax geometry and b) Multi-View triangle parallax geometry

2.3 Spatial 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 [Kakimoto 2005, 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 3b. 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 masks as shown in Figure 6. This mask is initialized once for the used N views of a specific display device during the render setup phase. After rendering the N views to N target images I_i , the final spatial image is combined by

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

For lenticular sheet displays the function $\text{Mask}(x, y)$ 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 6b.

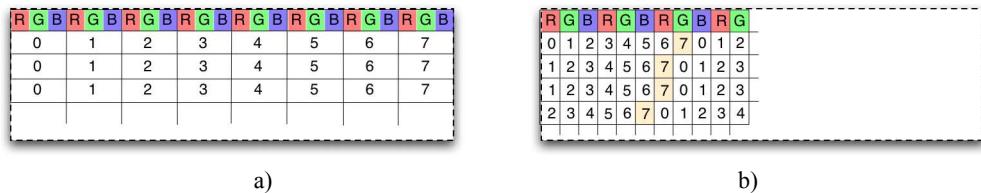


Figure 6. Interleaved Multi-view selection masks for a) lenticular sheets and b) wavelength-selective filter arrays ($N=8$)

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. To improve performance, intermediate images can be rendered at sub-resolution and restored to full-resolution during compositing 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 method which requires rendering the geometry 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.

3. SINGLE-PASS MULTI-VIEW RENDERING

3.1 Overview

For multi-view rendering our algorithm works significantly different from prior methods. In one single rendering pass we generate and illuminate multiple views directly on the GPU according to the multi-view parallax geometry illustrated in Figures 4 and 5. Based on the idea of covering the multi-view projections of a geometric primitive by an enlarged quad (Section 3.2), we use primitive dependent per-pixel ray intersection tests (Section 3.3) to determine the view-accurate projection of a rendering primitive to image-space. The view i of each pixel or sub-pixel is calculated by employing the display mask (Section 3.4). Finally Section 3.5 explains the details of our implementation for volume rendering, point splats and triangle primitives.

3.2 Multi-View Projection

In the context of a general multi-view configuration, i.e. the geometry is not coinciding with the focal plane (see also Figure 4 and 5), a geometric primitive projects differently for the different viewpoints as illustrated in Figure 7. Single-pass multi-view rendering requires to calculate a view per-pixel or sub-pixel, but access to the scene geometry on a per-pixel level is unfeasible. Therefore, we must cover all projections for multiple views, so the view calculation can be performed on a per pixel-basis during rasterization. This can be achieved by rasterizing an extended quadrilateral in image-space. As the views are only horizontally separated (Section 3), a wide quad $Q = (q^A, q^B, q^C, q^D)$ covering all possible projections of a geometric primitive can be drawn as shown in Figure 7a.

Given the maximal view eccentricity vd over all views, we can see that we need to find the minimal and maximal projection points r and s in Figure 7b. The covering quad has to be enlarged horizontally to cover this projection. Vertically, the quad only needs to cover the height of the perspectively projected primitive. Therefore, for a triangle primitive we have the quad corners given by:

$$q.x = \text{MIN}[(p_i.x - vd) \cdot fd / p_i.z] \mid \text{MAX}[(p_i.x + vd) \cdot fd / p_i.z]$$

$$q.y = \text{MIN}[p_i.y \cdot fd / p_i.z] \mid \text{MAX}[p_i.y \cdot fd / p_i.z]$$

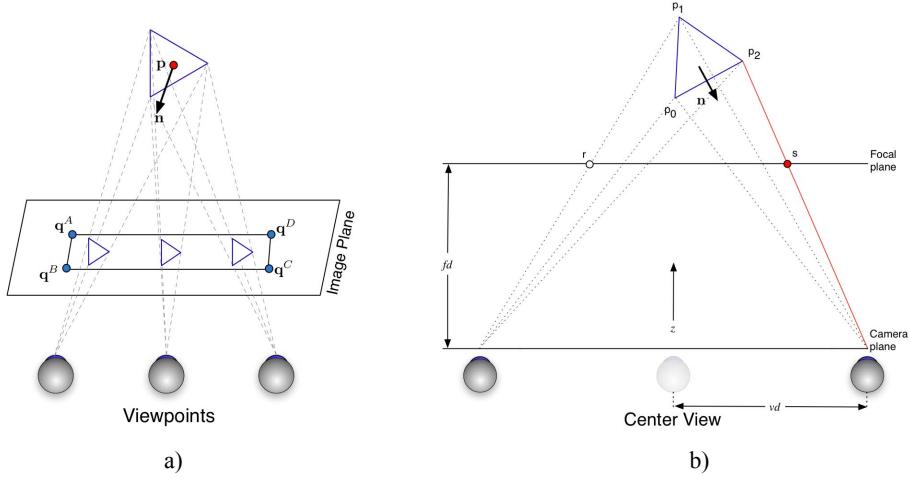


Figure 7. Multi-view dependent projection for a triangle geometric primitive

$$q.z = fd$$

The perspective accurate projection to the image-space is achieved primitive dependent by per-pixel ray intersection tests, that can be efficiently implemented in fragment shaders.

In multi-view volume rendering the proxy geometry is parallel to the image and camera plane (see Figures 4 and 8a). Moreover, the entire volume dataset is accessible on a pixel-level in the shader. This simplifies the approach significantly for multi-view volume rendering. Multiple views can be rendered in a single pass without the need to cover multiple projections, so the proxy geometry is left unchanged in the case of 3D texture based volume rendering.

3.3 Primitive Dependent Ray Intersection

Primitive dependent ray intersection tests are performed individually for each pixel or sub-pixel to find the accurate primitive-plane to image projection. We show exemplary the ray intersection tests for volume rendering (based on a viewport aligned proxy geometry), point splats and triangle primitives.

3.3.1 Ray-Volume Intersection

As illustrated in Figure 8a the proxy geometry for 3D-texture based volume rendering is parallel to the image plane, see also Section 3.5.1. Therefore multiple views can be calculated directly from the center view perspective projection using the offset

$$off_i = \frac{vd_i \cdot (d - fd)}{fd}, \quad (1)$$

where vd_i is the camera offset of view i and d is the distance of the visible point from the camera plane (see also Figure 4).

3.3.2 Ray-Splat Plane Intersection

Given a point sample p , its normal n , viewpoint vp and a pixel in image-space with coordinates $x = (x, y, fd)$ its corresponding projection \hat{x} on the splat plane (Figure 8b) is implicitly determined by $n \cdot (\hat{x} - p) = 0$ and $\hat{x} = vp + l \cdot (x - vp)$, hence

$$\hat{x} = vp + \frac{n \cdot (p - vp)}{n \cdot (x - vp)} \cdot (x - vp) \quad (2)$$

The ray $x - vp$ from a viewpoint vp through an image-plane pixel x is intersected directly with the splat plane according to Equation 2. The splat plane intersection \hat{x} then defines the distance as $\omega = |\hat{x} - p|$ that is used to decide if the intersection point is located inside the circular point splat or not.

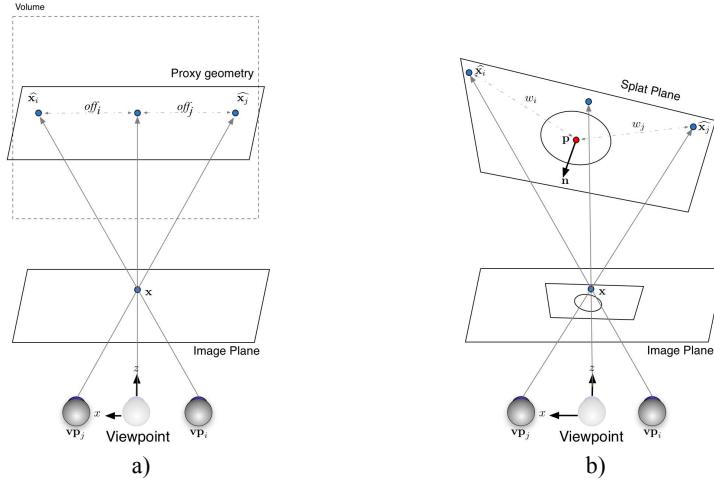


Figure 8. a) Ray-Volume Intersection and b) Ray-Splat plane intersection

3.3.3 Ray-Triangle Intersection

A point x on a triangle can be defined by its barycentric coordinates $x(u, v) = (1-u-v)*p_0 + u*p_1 + v*p_2$ where p_0, p_1, p_2 are the vertices of the triangle. The point x is located on the triangle if u and v fulfill the conditions $u \geq 0, v \geq 0$ and $u+v \leq 1$.

The intersection ray is defined by $\hat{x} = vp + l \cdot (x - vp)$ so the intersection between the ray and the triangle is given by $\hat{x} = (1-u-v) * p_0 + u * p_1 + v * p_2$. If there is a triplet (l, u, v) that satisfies the previous equation, and complies with the restrictions for u and v , then the ray intersects the triangle. The ray-triangle intersection can be fast and efficiently determined as described by [Möller and Trumbore 1997] solving the linear system:

$$[vp - x, p_1 - p_0, p_2 - p_0] \begin{bmatrix} l \\ u \\ v \end{bmatrix} = vp - p_0 \quad (3)$$

They interpret the linear system geometrically as a series of transformations (see Figure 9 with $M = [-(vp-x), p_1-p_0, p_2-p_0]$). First the triangle is translated to the origin and then transformed to a unit triangle in the y - z plane with the ray direction aligned with x .

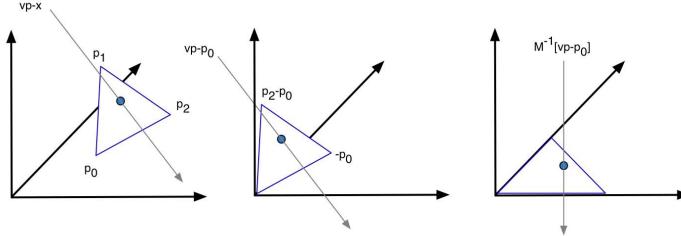


Figure 9. Geometric interpretation of Möller and Trumbore's triangle intersection algorithm

3.4 Employing Display Mask Map

According to the multi-view masks illustrated in Figure 6, there is one view $v = \text{Mask}(x, y)$ or three views $v_{\text{RGB}} = \text{Mask}_{\text{RGB}}(x, y)$ to consider for each pixel (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 pixel on the screen we need to determine the views for the pixels physical screen position (x, y) . The wavelength-selective filter mask in Figure 6b has a reoccurring pattern of size 8×12 . Depending on the physical screen position (x, y) , which is available in the fragment shader, we can calculate the x_{offset} and y_{offset} inside this mask by Equations 4 and 5, which refer to the mask index of the first sub-pixel component (R).

$$x_{\text{offset}} = (3 \cdot x) \% 8 \quad (4)$$

$$y_{\text{offset}} = (\text{screen_height_in_pixels} - y) \% 12 \quad (5)$$

The first view v_R of a fragment is computed by Equation 6 which considers the diagonal shifts of the wavelength-selective filter mask from Figure 6b. The other views are $v_G = (v_R + 1) \% 8$ and $v_B = (v_R + 2) \% 8$.

$$v_R = (\lfloor \frac{1 + yoffset + \lfloor \frac{yoffset}{3} \rfloor}{2} \rfloor + xoffset) \% 8 \quad (6)$$

3.5 Implementation

3.5.1 Volume Rendering

Our multi-view DVR algorithm is based on hardware accelerated 3D texture mapping [Cabral et al 1994, Gelder and Kim 1996, Meissner et al 1999]. We make use of pre-calculated voxel gradients which allow the use of high quality gradient estimation filters like the Sobel 3D operator [Jain 1989], and compared to on-the-fly gradient estimation in the shader [Engel et al 2006] achieve higher rendering speeds. Pre-calculated gradients come at the expense of increased memory cost. However, we store the volume data into one 3D RGB α -texture with gradient and scalar values in the RGB and α channels respectively. To improve visual quality, classification of the scalars is done by pre-integration as proposed in [Engel et al 2001]. The 2D pre-integration table is calculated and updated from the transfer function and kept in another 2D RGB α -texture.

Hardware accelerated 3D texture based volume rendering is performed by defining a proxy geometry e.g. a stack of viewport-aligned quadrilaterals (Figure 10a), with each quad having the size of the volume diagonal. Interpolated texture coordinates on the proxy geometry are used to sample the volume, and to extract scalar as well as gradient values. Gradients and scalars from the 3D texture are trilinearly interpolated. The number of proxy quads defines the volume sampling in z -dimension and is usually set to the volume resolution if pre-integration is used. While increasing the number of quads improves image quality, fewer quads result in higher frame rates.

The stack of quadrilaterals is set parallel to the viewport and camera plane (Figure 4). Rotation of the volume is achieved by reorienting the 3D texture (Figure 10b). To avoid excessive texture lookups and fragment processing, 3D volume aligned clipping planes are specified such that only the clipped quads are processed (Figure 10c).

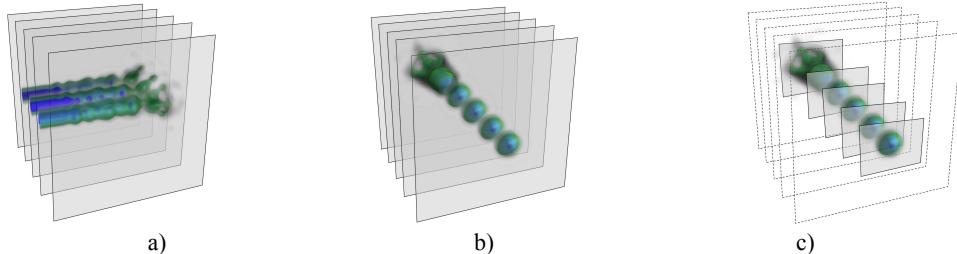


Figure 10. 3D volume texture mapping. a) Quadrilateral proxy geometry. b) Rotated 3D volume with viewport aligned quads. c) Proxy geometry clipped to rotated 3D volume

The proxy geometry is rasterized as clipped quadrilaterals on screen. The 3D texture volume is sampled for each generated fragment for two scalar values and gradients. The scalar values define a linear segment used to reference into the pre-integration table. Pre-integration reduces artifacts caused by high frequencies in the transfer function which otherwise require higher sampling. Using the volume gradients and lighting parameters, for each fragment (x,y) of every slice k the lighting equation is evaluated yielding a color $C_k(x,y)$. Rendering the slices back-to-front ($k=0...K$), the direct volume rendering integral is incrementally approximated by

$\widehat{C}_k(x, y) = C_k(x, y) + (1 - \alpha_k(x, y)) \cdot \widehat{C}_{k-1}(x, y)$, where $\widehat{C}_{k<0}(x, y)$ is the background color. Colors must be pre-multiplied with opacity α which is the case for pre-integration.

A standard vertex shader used for pre-integration [Engel et al 2006], computes the clipped vertex position (1), transformed texture coordinates (2, 8) and the coordinates of the linear segment used for pre-integration (2-8). For multi-view volume rendering we additionally compute the viewport-parallel direction of the parallax offset off_i in texture space (9). We also have to pass the eye-space distance d – from the proxy quad to the camera plane – to the fragment shader (10), for calculating the multi-view parallax offsets as in Equation 1.

```

void vertex_shader (void) {
1   gl_Position      = ftransform();
2   gl_TexCoord[0]    = gl_TextureMatrixInverse[0] * gl_Vertex;
3   vPosition        = gl_ModelViewMatrixInverse * (0, 0, 0, 1);
4   vDir             = normalize(gl_ModelViewMatrixInverse * (0, 0, -1, 1));
5   eyeToVert        = normalize(gl_Vertex - vPosition);
6   backVert         = (1, 1, 1, 1);
7   backVert         = gl_Vertex - eyeToVert * (sliceDistance / dot(vDir, eyeToVert));
8   gl_TexCoord[1]    = gl_TextureMatrixInverse[0] * backVert;
9   gl_TexCoord[3]    = gl_TextureMatrixInverse[0] * (1, 0, 0, 0);
10  d                = (gl_ModelViewMatrix * gl_Vertex).z;
}

```

For the fragment shader shown below, the 3D RGB α volume texture, the 2D RGB α pre-integration table as well as the multi-view configuration parameters (focal distance, camera offset) are passed as uniform variables from the application. According to the multi-view masks illustrated in Figure 6 and Section 3.4 the views are determined (1-3). After knowing the view for each sub-pixel of the fragment, the multi-view offsets off_i can be calculated based on Equation 1. The targeted auto-stereoscopic display has $N=8$ possible views. Hence vd_i translates for a given view i at a fixed intraocular distance $iocd$ to $vd_i = (i - 3.5) \cdot iocd$, and the

$$off_i = \frac{(i-3.5) \cdot iocd \cdot (d-fd)}{fd}$$

final equation for calculating the offset for a particular view is thereby with the intraocular distance $iocd$ and focal distance fd given by the application (6). For each view the texture coordinates of the pre-integrated segments are calculated by an offset along the parallax direction (7, 8), and the 3D texture lookups provide two gradient and scalar values (9, 10). The scalar values index into the pre-integration table for the corresponding color and opacity values (11).

Lighting uses the averaged and normalized volume gradients (12) as surface normals in the Blinn-Phong shading model $C = I_a \cdot k_a + I_d \cdot k_d(\vec{N} \cdot \vec{L}) + I_s \cdot k_s(\vec{N} \cdot \vec{H})^{sh}$, with the ambient, diffuse and specular light intensity $I_{a,d,s}$, reflection coefficients $k_{a,d,s}$ and shininess factor sh . All three sub-pixel views are shaded independently depending on the direction of the surface normal (13-16). Note that current GPUs do not support a different α value per color channel for α -blending. Thus at this time we average the fragment's α over the three contributing views. Nevertheless, comparing to a multi-pass multi-view implementation not exhibiting this restriction, no visual artifacts are noticeable.

The code below shows a simplified sub-pixel resolution fragment shader implementing our single-pass multi-view volume rendering approach.

```

void fragment_shader (void) {
1   xoffset = mod(3. * gl_FragCoord.x, 8.);
2   yoffset = mod(1200. - gl_FragCoord.y, 12.);
3   view.r = mod(floor((1. + yoffset + floor(yoffset / 3.)) / 2.) + xoffset, 8.);

6   off     = (view - 3.5) * iocd * (d - fd) / fd;

7   offsetSF [0..2] = gl_TexCoord[0].xyz + off.x.z * gl_TexCoord[3].xyz;
8   offsetSB [0..2] = gl_TexCoord[1].xyz + off.x.z * gl_TexCoord[3].xyz;
9   lookupSF [0..2] = texture3D(volume, offsetSF[0..2]);
10  lookupSB [0..2] = texture3D(volume, offsetSB[0..2]);

11 c      [0..2] = texture2D(preInt, vec2(lookupSF[0..2].a, lookupSB[0..2].a));
12 normal [0..2] = normalize(2. * lookupSF[0..2].rgb - 1. + 2. * lookupSB[0..2].rgb - 1.);

13 diffuse      = max(gl_LightSource[0].position.xyz * mat3(normal[0], normal[1], normal[2]),
0.0);
14 specular     = max(gl_LightSource[0].halfVector.xyz * mat3(normal[0], normal[1], normal[2]),
0.0);
15 specular.r.b = pow(specular.r.b, sh);

16 gl_FragColor = ( gl_LightSource[0].ambient    * (c[0].r, c[1].g, c[2].b) +
                    gl_LightSource[0].diffuse     * (c[0].r, c[1].g, c[2].b) * diffuse +
                    gl_LightSource[0].specular   * (c[0].r, c[1].g, c[2].b) * specular,
(c[0].a + c[1].a + c[2].a) / 3. );
}

```

3.5.2 Point Splatting

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.

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 2006a, Zhang and Pajarola 2006b]. 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.

The multi-view splatting algorithm follows the main steps of this 1+1 pass point based rendering (PBR) algorithm: dividing the points into multiple groups in a preprocessing stage, and then 1) rendering groups 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.

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 3.2, an enlarged quad covering splat projections of all views is rendered. We choose to calculate the quad corners in a vertex shader according to the Equations in Section 3.2. 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 to calculate the corners of the extended quad.

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

Geometry Pass:

```

1 turn on z-test and z-update;
2 foreach group  $S_k$  do
3   clear z-depth and color of texture  $I_k$ ;
4   render group  $S_k$  to texture  $I_k$ ;
5   foreach  $s_i$  element  $S_k$  do
6     calculate the corresponding corners of the quad
     according to Equation in Section 3.2;
7     transform and project the corners of the quad;

```



```

8   foreach generated fragment f element  $I_k$  do
9     determine sub-pixel views corresponding to current
     fragment;
10    perform ray-splat intersection calculations;
11    calculate colors, kernel weights and depth values;
12    output averaged z-depth  $c_d(f)_k$ ;
13    pack colors and kernel weights, and output them to
     texture  $I_k$ ;
14  endforeach
15 endforeach
16 endforeach
```

3.5.3 Triangle Meshes

Our multi-view algorithm for rendering triangle meshes applies the same multi-view projection principle as we use for multi-view point splatting to triangles. A standard algorithm renders the triangle mesh whereas each triangle is rendered as a quad primitive. Therefor we input the last triangle vertex twice. An index is assigned to each of the vertices, identifying the quad corners. The initial triangle coordinates are attached to each vertex as multi-texture coordinates. This allows us to enlarge the quad properly in the vertex shader and to calculate the exact triangle projection based on the intersection test introduced in Section 3.3.3.

3.6 Complex Output

Compared to previous methods, our multi-view algorithm is more complex due to the fact, that we have to handle multiple views simultaneously instead of one single view. For each view, we need to store color information and depth values. Different output strategies are applied for volume rendering, point splatting and triangle meshes.

Image Compositing Pass:

```

1 foreach f element I do
2    $c_{rgb}(f) = 0$ ;
3    $w_{rgb}(f) = 0$ ;
4    $d = \min_k(c_d(f)_k)$ ;
5   for  $k=0$  to  $K-1$  do
6     if  $c_d(f)_k \leq d + \epsilon$  then
7       unpack color values to
 $c_{rgb}(f)_k$ ;
8     unpack weight values to
 $w_{rgb}(f)_k$ ;
9      $c_{rgb}(f) = c_{rgb}(f) + c_{rgb}(f)_k$ ;
10     $w_{rgb}(f) = w_{rgb}(f) + w_{rgb}(f)_k$ ;
11  endif
12 endfor
13  $c_{rgb}(f) = c_{rgb}(f) / w_{rgb}(f)$ ;
14 endforeach
```

For volume rendering the z -depth is the same for all views inside a single fragment. Hence only the color values of maximal three different views need to be combined into one output color. While each fragment color component is set to the corresponding view's R-, G- or B-component, the alpha values of all views are averaged and written to the alpha channel.

Rendering point splats, actually requires an averaged depth value for multiple views which we employed in our implementation because the ε - z operation in the composting pass renders a high precision depth value unnecessary. But even with the averaged depth value used, three different color values need to be forwarded from the geometry processing pass to the image composition pass. We use a 32-bit floating point texture to store the output from the geometry processing. The high precision texture allows us to use packing operations to pack the colors as two half floating values into a single 32-bit floating value and then to unpack them in the image compositing pass.

Correctly rendering triangle meshes requires an exact depth value. Thus combining the depth values from different views by averaging causes artifacts that are minimized when using the minimal depth instead.

4. RESULTS

We have implemented our multi-view rendering methods in OpenGL using OpenGL Shading Language 1.1. The presented results were generated on a 2.66GHz MacPro with a ATI Radeon X1900 graphics card supported by 512MB. The targeted multi-view auto-stereoscopic display uses a wavelength-selective filter array providing 8 views and has a resolution of 1900x1200.

Our single-pass multi-view rendering algorithm (GPU MV) is compared to a standard N -pass (8 MV) and to a baseline standard single-view (SV) implementation. To analyze the basic behavior, all algorithms use the same code base. For efficiency, the standard multi-view algorithms (8 MV) use a render-to-texture approach, and multi-texturing of the N view images for compositing and masking of the final spatial image.

The GPU MV rendering algorithm has to compute three different views – one for each RGB component – per processed fragment. Consequently, the performance of GPU MV can be expected to be significantly better for lenticular sheet displays with only a single view per pixel.

In Chart 1 we report frame rates for generating ($N=8$)-view spatial images at 512×512 resolution for different standard volume data sets. As expected, the standard 8 MV algorithm suffers significantly from the 8 rendering passes it has to perform. The expected performance ratio of 8:1 between SV and 8 MV roughly holds, and GPU MV improves over 8 MV by a factor of $\approx 2\text{-}4$. Moreover, we can see that the theoretical ratio of 3:1 between SV and GPU MV holds for medium data sets and, that our algorithm performs increasingly better for larger data (Chart 1b).

SINGLE-PASS MULTI-VIEW RENDERING

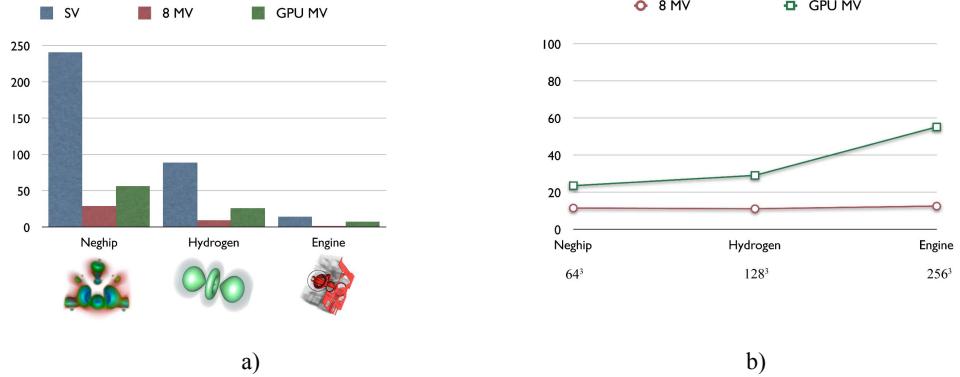


Chart 1. Multi-View Volume Rendering (512×512 window resolution, N=8 views)
a) Performance comparison in fps and b) Comparison in percentage of single-view performance for different data sets.

Our multi-view point splatting implementation improves in the multi-pass rendering by having at least 50% of the single-view performance. Chart 2 shows the performance comparison for different point data sets. Our multi-view implementation performs better with increasing data set sizes. Single-view PBR shows a strong dependance on the data set size while the multi-view performance reduces much less. As can be seen in Chart 2b, 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 MV multi-view splatting algorithm achieves almost 100% of the performance of single-view rendering for the biggest data set as shown in Chart 2b, notably generating 8 different views simultaneously.

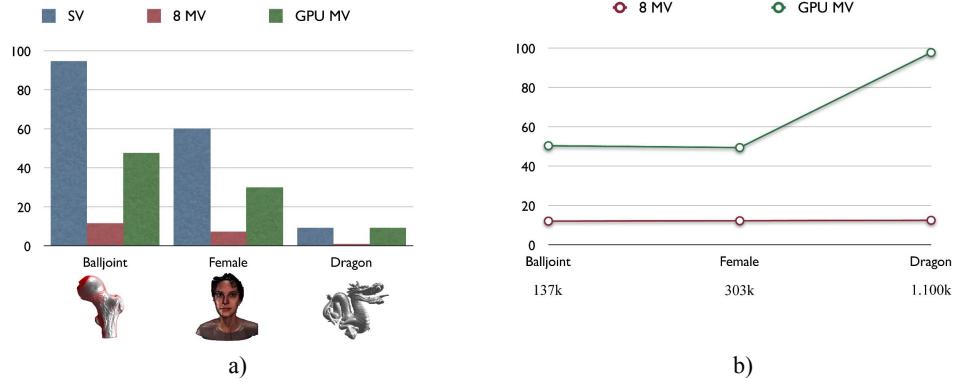


Chart 2. Multi-View Splatting (512×512 window resolution, N=8 views)
a) Performance comparison in fps and b) Comparison in percentage of single-view performance for different data sets.

In Chart 3 the results for multi-view triangle rendering are presented, for different triangle mesh data sets. The multi-view single-pass triangle mesh rendering algorithm (GPU MV) does not improve very much over the multi-pass standard triangle rendering method (8 MV). The main reason for this can be found in two observations. First, the basic triangle rendering

pipeline is highly optimized and any processing of geometry deviating from the standard fast-path, i.e. rendering quads instead of triangles, will immediately incur a significant performance penalty. Second, in the ray-triangle intersection fragment shader a high-percentage of fragments from the rasterized covering quad will be discarded as they eventually do not contribute to any visible parts of the triangle in any view. However, the discard fragment shader instruction does not in fact cause an early termination of the shader execution on the currently tested GPU. Hence efficiency improvements from fragment discards will only be exploited in future GPUs.

For comparison we additionally provide the results of our multi-view triangle rendering algorithm for rendering in the immediate mode of OpenGL (Chart 4). Being less optimized, the observed rendering performance of our multi-view algorithm is inside the expected range of $\geq 1/2$ of the single-view performance.

It is important to note, that the performance of our multi-view method 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. For multi-view volume rendering this results in a larger offset which increases the volume access time. Nevertheless, the number of generated fragments as well as the offset for objects placed far away from the focal- and camera plane is limited by the maximum view eccentricity vd and for point splats and triangles by the projection of the geometric primitive according to Equation in Section 3.2.

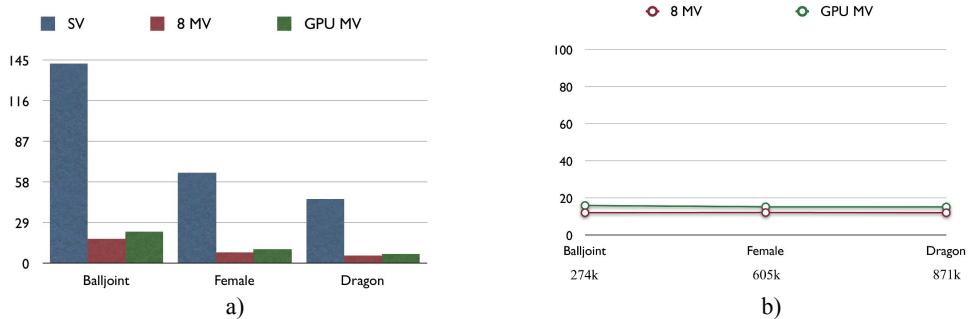


Chart 3. Multi-View Triangle Rendering with Display List (512 × 512 window resolution, N=8 views)
a) Performance comparison in fps and b) Comparison in percentage of single-view performance for different data sets.

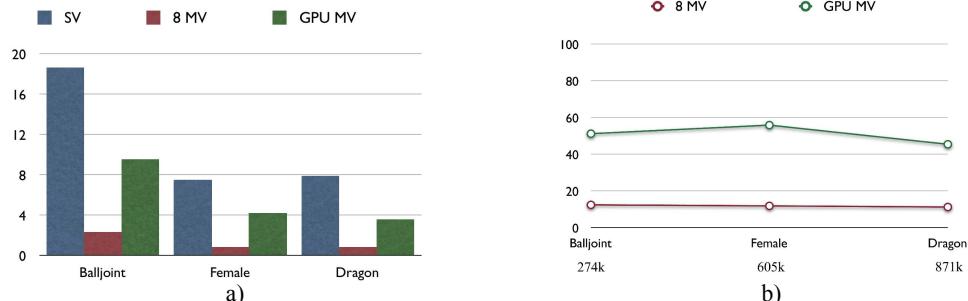


Chart 4. Multi-View Triangle Rendering, OpenGL Immediate Mode (512 × 512 window resolution, N=8 views)
a) Performance comparison in fps and b) Comparison in percentage of single-view performance for different data sets.

5. CONCLUSION

We presented a novel multi-view rendering method that exploits the programmability of modern GPUs to calculate multiple views directly on the GPU, in a single geometry rendering-pass. This multi-view rendering method is proposed for volume rendering, point splatting and triangle meshes. The implementation of our multi-view algorithms is compared with the performance of a standard multi-view and single-view visualization. The experimental results for volume rendering and point splatting show a significant performance advantage compared to the standard multi-view multi-pass rendering. In particular, our multi-view implementation can render N -views with more than $\frac{1}{N}$ of the single-view performance compared to approximately $1/N$ 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 data sets as well as for single-stereo. Though the performance improvement for triangle meshes is minor, 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. Currently, the triangle-processing optimized hardware pipeline causes significant overhead for our multi-view rendering algorithm. However, with unified shaders as well as true fragment-discard implementation in the fragment shaders our multi-view triangle rendering solution this situation will change. Our multi-view triangle rendering method will dramatically benefit from these advanced shaders.

ACKNOWLEDGMENT

We would like to thank Newsight technology for their technical support for the 3D display device as well as the Stanford 3D Scanning Repository, Cyberware and VolVis.org for providing the 3D 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

- Botsch, M. and Kobbelt, L., 2003. High quality point-based rendering on modern GPUs. *Proceedings Pacific Graphics 2003, Computer Society Press*, pp. 335-343.
- Cabral, B. et al, 1994. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *Proceedings ACM/IEEE Symposium on Volume Visualization*. Tysons Corner, USA, pp. 91-98.
- Dodgson, N. A., 2002. Analysis of viewing zone of multi-view autostereoscopic displays. *Proceedings of SPIE*, Vol. 4660, The International Society for Optical Engineering, pp. 254-265.
- Dodgson, N. A., 2005. Autostereoscopic 3D displays. *IEEE Computer*, Vol. 38, No. 8, pp. 31-36.
- Ebert, D. et al, 1996. Two-handed interactive stereoscopic visualization. *Proceedings IEEE Visualization 96*. San Francisco, USA, pp. 205-210.
- Engel, K. et al, 2001. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on graphics hardware*. Los Angeles, USA, pp. 9-16.
- Engel, K. et al, 2006. *Real-time volume graphics*. A K Peters, Wellesley, USA.

- Gelder, A. V. and Kim, K., 1996. Direct volume rendering with shading via three-dimensional textures. *Proceedings ACM/IEEE Symposium on Volume Visualization*, San Francisco, USA, pp. 23-30.
- Hübner, T. et al, 2006. Multi-View Point Splatting. *Proceedings Conference on Computer Graphics and Interactive Techniques in Australasia and South-East Asia (GRAPHITE)*, Kuala Lumpur, Malaysia, pp. 285-294, 499.
- Hübner, T. and Pajarola, R. 2007. Single-Pass Multi-View Volume Rendering. *Proceedings IADIS Multi Conference on Computer Science and Information Systems*, Lisbon, Portugal, pp. 50-58.
- Jain, A. K., 1989. *Fundamentals of Digital Image Processing*. Prentice Hall, New Jersey, USA.
- Kakimoto, M., 2005, Interactive Stereo Library (ISL). *Joint Meeting on 3D Technology*. London, England, <http://www.sgi.co.jp/solutions/visualization/isl>.
- Lee, Y. G. and Ra, J. B., 2005, Reduction of the distortion due to non-ideal lens alignment. *Proceedings of SPIE*, Vol.5664, pp. 506-516.
- McAllister, D. et al, 1993, *Stereo Computer Graphics and other True 3D Technologies*. Princeton University Press, Princeton, USA.
- McAllister, D., 2002, *Encyclopedia of Imaging Science and Technology*. John Wiley & Sons, Hoboken, USA.
- Meissner, M. et al, 1999, Enabling classification and shading for 3D texture mapping based volume rendering using OpenGL and extensions. *Proceedings IEEE Visualization 99*. San Francisco, USA, pp. 207-214.
- Miniel, S. et al, 2004, 3D functional and anatomical data visualization on auto-stereoscopic display. *Proceedings EuroPACS-MIR*, Trieste, Italy, pp. 375-378.
- Möller, T. and Trumbore, B., 1997, Fast, minimum storage ray-triangle intersection. *Journal of Graphic Tools*, Vol. 2, No. 1, pp. 21-28.
- Pajarola, R. et al, 2004, Confetti: Object-space point blending and plattting. *IEEE Transactions on Visualization and Computer Graphics 10*, Vol. 5, pp. 598-608.
- Pfister, H. et al, 2000, Surfels: Surface elements as rendering primitives. *Proceedings ACM SIGGRAPH*. New Orleans, USA, pp. 335-342.
- Portoni, L. et al, 2000, Real-time auto-stereoscopic visualization of 3D medical images. *Proceedings of SPIE*, Vol. 3976, Medical Imaging 2000: Image Display and Visualization, pp. 37-44.
- Ren, L. et al, 2002, Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. *Proceedings EUROGRAPHICS*. Saarbrücken, Germany, pp. 461-470.
- Schmidt, A. and Grasnick, A., 2002, Multiviewpoint autostereoscopic displays from 4D-Vision GmbH. *Proceedings of SPIE*, Vol. 4660, The International Society for Optical Engineering, pp. 212-221.
- Zhang, Y. and Pajarola, R., 2006, GPU-accelerated transparent point-based rendering. *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*, pp. 37-48
- Zwicker, M. et al, 2001, Surface splatting. *In Proceedings ACM SIGGRAPH*, Los Angeles, USA, pp. 371-378