Hardware-Assisted Visibility Ordering for Point-Based and Volume Rendering

Christian Hofsetz
Ciências Exatas e Tecnológicas
Universidade do Vale do Rio dos Sinos
chofsetz@acm.org

Nelson Max
University of California, Davis
Visualization and Graphics Research Group
max2@llnl.gov

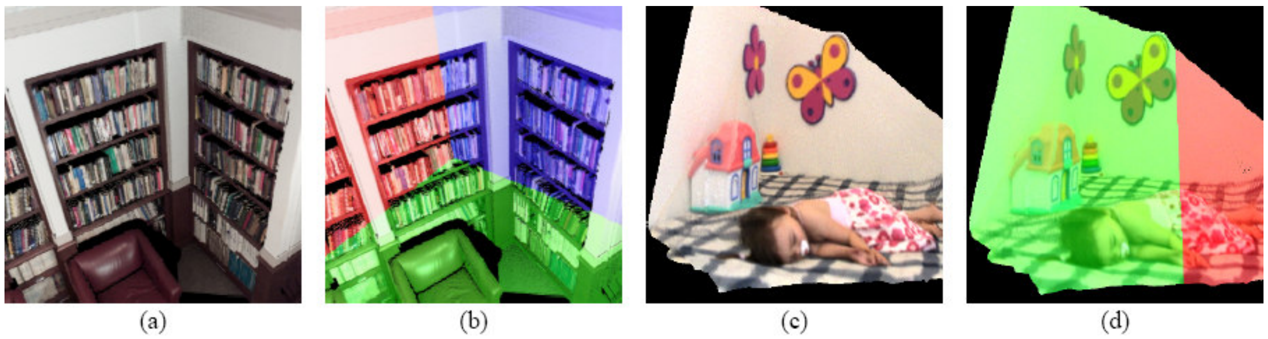


Figure 1. Example of the application of our technique to two different rendering techniques. We show the images in pairs to emphasize the absence of artifacts where the boundaries of our axes lists meet. (a) and (b) classic splatting; (c) and (d) image-based rendering of range data with estimated depth uncertainty. In (b) and (d), Red = X axis, Blue = Z axis, Green = Y axis.

Abstract

This paper presents a method to accelerate algorithms that need a correct and complete visibility ordering of their data for rendering. The technique works by first pre-sorting primitives using three lists - one for each axis, and then combining them using graphics hardware, by either clipping the projected primitives while rendering, according to the current list being processed, or by rendering each list to a texture and merging the textures in the end. We show that our algorithm works by applying it to the splatting technique using several types of rendering, including classic splatting and volume rendering.

1. Introduction

One of the problems in point-based computer graphics is how to spread the contributions of the 2D projection from a single 3D point. The projection of a 3D point in the image space occupies only one pixel. The rendering algorithm somehow has to fill in the gaps in between pixels in order to avoid holes in the image. Moreover, even if the projected point cloud results in several points per pixel (so all their projections do not cause any holes), the high density of the points may still cause the problem of aliasing.

The most common method of rendering point clouds is to increase the size of each projected point to multiple pixels [1] [2] [3]. The contribution of the projected primitive has to be blended with the contribution of its neighbors. This technique is called *splatting*. Now, in order to blend the primitives correctly, they must be processed in a specific order in relation to the camera viewpoint (called *visibility ordering*). In fact, some primitives may not even be used for blending if the resulting blending of the foreground objects that lie in its projected area is completely opaque.

This paper presents a new technique that searches *all* the data in linear time during rendering, using three pre-sorted lists. Our method can be used for any rendering algorithm that needs visibility ordering. Since it allows the visualization of inner structure, it also can be used for volume rendering.

2. Previous and Related Work

The splatting technique projects object primitives to the screen. If the original object primitive is only a point or a voxel, its projected contribution is spread over an area for hole-filling and anti-aliasing purposes (i.e., the point is *splatted*). This area is called a *kernel*, a *footprint*, or a *splat*, which can be defined in object-space (for instance, by centering a fixed-size disk on the 3D point) or in image-space. Each point of the kernel (or its projection) on the screen has color, and blending information, usually in the form of opacity.

If more than one kernel is projected to the same pixel, a composition scheme is needed, which often depends on the ordering of the splats. Occlusion is dealt with by either (1) searching through the objects in a specific order defined by the viewpoint location, or by (2) identifying which objects are actually visible before blending. Approach (2) is usually much faster because it only selects the frontmost splats for rendering.

Both approaches may still cause holes in the final image. The solution for this problem is to make the footprint as large as necessary to avoid these holes. This is not a simple task, since we may not have all the information necessary in order to fill the holes (for example, when we have range data acquired from photographs). Also, we may fill holes that should not be filled - i.e., they are part of the scene.

Another way to avoid holes in the rendered image is to ensure that there are enough point samples to match the resolution of all desired virtual views, i.e., if the scene is adequately sampled as shown by Grossman and Dally [4]. Holes can then be filled by blending pixels from lower resolution approximations of the image.

In order to speed up rendering, it is possible to construct a hierarchical representation of the data. Laur and Hanrahan proposed in [5] a hierarchical splatting algorithm, where the volume data is represented as a pyramid. Each level of the pyramid stores a different size footprint. During rendering, more than one level of the pyramid may be used in order to include greater detail from points closer to the viewpoint.

Many image-based rendering methods use images with depth (or range images) as primitives for rendering. Images with depth may contain more information about the data than just the color and 3D position. The visibility ordering is achieved by exploiting this information, using specific data structures and search orders. For example, it is very common to organize those images as a 3D structure called a layered depth image (LDI) [6] [7]. LDIs can be organized into hierarchical representations [8] to improve rendering performance.

The more recent surfel technique [9] [10], uses a hardware-assisted two-pass solution for this problem. First, the visible pixels are determined by projecting opaque polygons for each point (or surfel) to the screen. In the second pass, the z-buffer will then select the frontmost projected splats, which are defined as object space Elliptical Weighted Average (EWA) resampling filters, and projected and rendered as textured quadrilaterals. The algorithm is implemented using programmable vertex and pixel shaders.

Splatting is also used in volume visualization algorithms. In the first implementation of the splatting technique for volume rendering [3], a front-to-back or back-to-front order is necessary in order to solve occlusion. Voxels that are closer to the image plane take precedence over farther voxels.

In order to avoid sorting, Westover [3] chooses the axis that is closest to perpendicular to the image plane and renders the slices of that axis in a front-to-back order. This is called *axis-aligned splatting*. It produces some artifacts when the selected axis changes. Mueller and Crawfis proposed a modification in the technique by slicing the volume with sheets parallel to the image plane [11], thus avoiding the artifacts.

Recently, Chen et al. in [12] implemented Westover's algorithm in hardware, using Elliptical Weighted Average volume splatting. Their algorithm attained interactive frame rates by loading the whole visible volume (i.e., the part with nonzero opacity) to the graphics memory. They were not able to remove the "popping" artifacts.

We have developed here a new artifact-free technique that maintains the visibility ordering needed for some applications, while being generic enough to be applied to several different point-based and volume rendering datasets.

3. Our Approach

A simple way to guarantee a back to front order of compositing is to form three lists of points, sorted along the X, Y, and Z axes. Then we choose the axis that is most closely aligned with the viewing ray (or its negative) and use that list in its sorted (or reverse sorted) order. There are three ways to choose the axis. (1) If this decision is made per frame, as in [3], based on the viewing ray at the image center, the whole image may appear to pop at the frame which switches to a new sorting order. (2) In a perspective view, the decision can be made per pixel ray, using clipping. This was first introduced in [13] and is generalized here for any point-based and volume rendering data. (It is the basic algorithm presented in section 5.) In this case, there is a moving set of transition lines across which the order changes with the changing view, creating smaller but still visible artifacts, as the switching region becomes 1-D in image space but also 1-D in time. (3) In this paper, we also propose a third method, which effectively blends the sorting order in a region near these transition lines, so that the artifacts are much less noticeable.

Note that unlike [11], we do not need to resample the volume. Therefore, the application of our technique to the real-time framework presented in [12] is straightforward.

4. Ellipsoidal and Elliptical Gaussian Kernels

The goal of the splatting technique is to reconstruct images from a point cloud. The original point cloud is created using some kind of sampling technique - either from synthetic or real data. The reconstruction is obtained by the blending of the splats from the point cloud. Each type of splat used may have several different shapes and blending properties. The use of splats affects the resulting aliasing and frequency of the image because they are actually *filtering* the data [14].

Filtering a signal means to *bandlimit* its frequency. The idea is to discard all the frequencies that we believe do not belong to the original signal or cannot be reconstructed. The filters that are most often used are *low-pass filters* - they cut off the copies of higher frequencies.

In this paper we are particularly interested in splats defined by a truncated Gaussian kernel. There are several advantages to using a Gaussian kernel as a filter. Gaussians are very efficient and easy to implement, and a small kernel is usually enough for our reconstruction.

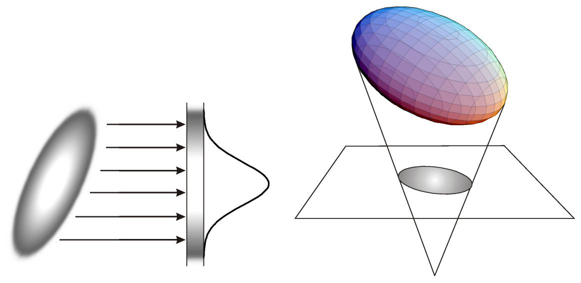
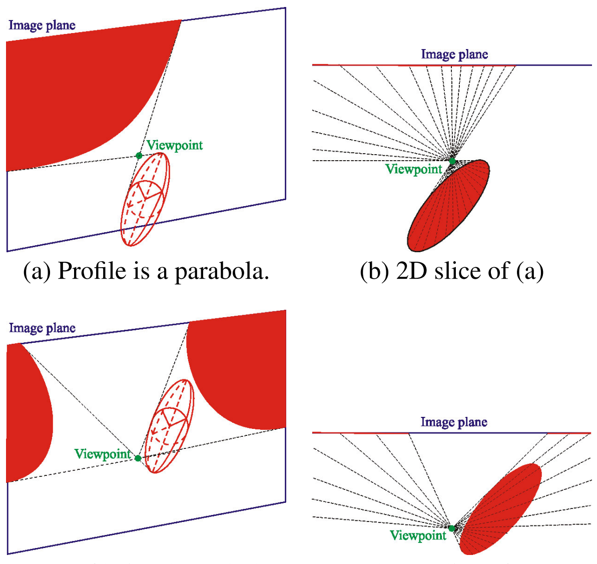

(a) Orthogonal projection (b) Perspective projection

Figure 2. (a) The orthogonal projection of an n-D Gaussian is an $(n-1)$-D Gaussian. (b) The perspective projection of an ellipsoid is an ellipse

Furthermore, for the purpose of using the framework presented in this paper for volume rendering, we have to select a filter that fits our 2D technique. In fact, it can be shown that in an orthogonal projection, the integral of a 3D Gaussian is a 2D Gaussian [15] (see Figure 2a). This is based on the Volume Splatting work of Zwicker in [16], and can be proved by integration along the projection direction.

In the perspective projection case, we are interested in the projection of the 3D ellipsoid onto the image plane. We know that the perspective projection of an ellipsoid is an ellipse. Figure 2b illustrates that the projection of an ellipsoid is a conic section [15].



(a) Profile is a parabola. (b) 2D slice of (a)

(c) Profile is a hyperbola. (d) 2D slice of (c)

Figure 3. Ellipsoid perspective projection may be a hyperbola or a parabola.

If the ellipsoid is entirely on one side of the plane through the viewpoint, the elliptical cone will intersect the image plane in a single, connected curve, which lies in a bounded region of the plane, so it must be an ellipse (or a circle). Notice that here we must assume that the ellipsoid is always in front of the image plane. Figure 3 illustrates that if our assumption does not hold, when the ellipsoid extends partly behind the viewer, the projected profile may be a hyperbola or a parabola.

Therefore, since the orthogonal projection of a 3D Gaussian is a 2D Gaussian, and an ellipsoid projection is an ellipse, we can approximate a 3D Gaussian by a 2D elliptical kernel. Thus, if we use a Gaussian kernel in our algorithms we can apply the same technique for volume rendering as well, even for perspective projection.

5. Our Algorithm

The application of our technique in any point-based rendering framework is straightforward. First, we create three sorted lists of all primitives, one for each axis X, Y, and Z. Then, for each list, we project all points to the image. In our implementation, we represent each projected primitive as an ellipse (it can be any shape, depending on the application). The ellipses are rendered as textured quadrilaterals. As the splats are added to the image, we accumulate their color and opacity contributions using standard alpha-blended composition. The contribution of each splat is stored in the alpha channel. Finally, the color channel is normalized by dividing it by the accumulated alpha.

If our data is volumetric, each primitive is represented as an ellipsoid, which is the support of our 3D Gaussian kernel. Then, based on the Volume Splatting work of Zwicker in [16], we project the ellipsoid onto the image plane. As seen in the previous section, each projection of an ellipsoid is an ellipse.

The pseudocode for this technique is:

1. Build three global sorted lists, one for each axis (X, Y and Z). For some data the sorting is not necessary (e.g., volume data in a structured grid). See section 6.

2. For each axis, and each primitive in back to front order on that axis

   (a) Calculate the elliptical projection of the primitive onto the image plane, obtaining the corners of a quadrilateral enclosing the projected ellipse.

   (b) Clip the quadrilateral, if necessary.

   (c) Calculate and composite the color and opacity of each pixel in the quadrilateral, using the texture.

3. Normalize the accumulated color, by dividing by the accumulated alpha.

4. Display the image.

Figure 4 illustrates our rendering pipeline. After the primitive is projected by the CPU, the rendering is finalized by the OpenGL engine, with the help of vertex and fragment programs.

Although the lists are searched using the correct visibility order of the primitives in step 2, which list we pick first does not matter. Step 2c is implemented as a fragment program. The fragment color and opacity are computed by multiplying the color from the primitive and the gaussian texture. If the alpha value of the fragment is less than or equal to 0.0001 then the fragment is killed. This is done to remove small contributions from the pipeline to avoid numerical errors. The final step of normalizing the color of the image (step 3) is also implemented as a fragment program.

5.1. Pre-Sorting

We maintain three lists of all ellipsoids sorted by the X, Y and Z components of each primitive. Each pixel in the image will receive contribution for only one of these lists.

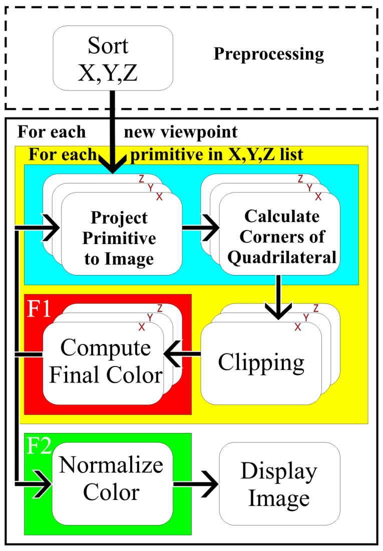


Figure 4. The rendering algorithm. Modules in F1 and F2 are executed by fragment programs.

We pick the axis list corresponding to the component of the viewing ray's direction vector with largest absolute value.

The list to be chosen depends on the X, Y and Z components of the virtual ray of that pixel. The choice is defined by the component with largest absolute value. The direction of searching the list is defined by the sign of this component, front-to-back if it is negative and back-to-front if it is positive.

A list is only searched if there is at least one viewing ray whose largest component X, Y or Z is the one that identifies the list.

For the purpose of rendering, we first project all primitives to the image plane (step 2a of the algorithm). During this step, we also identify which lists (X, Y and/or Z) we have to process to project each primitive. A few primitives may have to be splatted two or three times.

Of course, each pixel should not receive the contribution of a splat more than once. The splat must be clipped accordingly, and this is done using OpenGL clipping planes.

5.2. Removing Artifacts

The method just described has some limitations that may cause artifacts for some models. The majority of the problems occur in volumetric datasets.

If the size of the projected splat is too large, if the splat overlaps several other splats, and if this happens near the limit of an axis region, the separation between the axes will be visible in the final image. However, for still images, we have only noticed this problem when we generated a synthetic volume with large ellipsoids. This problem may also happen for datasets with elliptical surface splats when we have a similar situation (i.e., large splats), although in this case the artifacts will be much less noticeable, since the frontmost implicit surface will be completely opaque. Similar artifacts may appear when we use a regular grid volumetric dataset, although they are barely noticeable - the lines will only be evident when we zoom in into the image (as in Figure 10b, which is a close-up view of Figure 9a). For animations using a moving viewpoint the artifacts are more noticeable, because they move coherently.

These artifacts are the result of the discrete nature of the algorithm. There is a discontinuity at the borders of the regions. To remove these artifacts we have to modify the algorithm for the purpose of making a smoothly blend transition between the sorted orders near the lines where they switch.

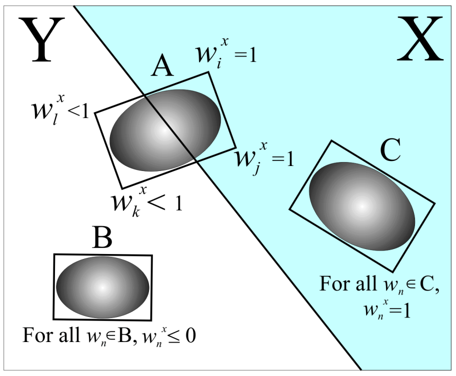

Y
X
A
$w_i^x = 1$
$w_l^x < 1$
$w_j^x = 1$
$w_k^x < 1$
C
B
For all $w_n \in$ C, $w_n^x = 1$
For all $w_n \in$ B, $w_n^x \leq 0$

Figure 5. Corner weights. X and Y indicate the regions of axes X and Y. If the list of axis X is being rendered, splats A and C are projected, and splat B is discarded.

The solution for this problem is twofold. First, we render each list to a hardware pixel buffer (pbuffer) (Figure 6). Second, for every list, we render each quadrilateral vertex i with an extra parameter w_i (a weight) at each corner i. Assuming that the viewpoint is at the origin, w_i is calculated per-vertex v_i:

$$w_i = ((v_i^{x,y,z}/v_i^{MAX}) - h)/(1 - h)$$

where $v_i^{x,y,z}$ is the component x, y or z of the viewing ray for vertex i and v_i^{MAX} is the largest component of the viewing ray for vertex i. The component to be chosen (x, y or z) is the same component as the current list being searched. For example, if we are processing the list of axis X, v_i^x is chosen for weighting. Now, h indicates how far into the

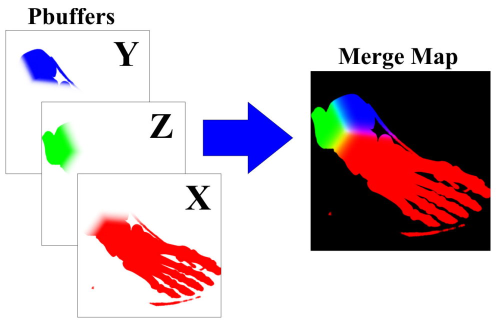


Figure 6. Merge map for $h = 0.85$. Each X, Y and Z list is rendered to a pbuffer texture. The pbuffers are merged in the last stage of the rendering. Areas of solid color indicate where no merge is necessary.

neighbor's region the algorithm is going to project splats for blending.

While searching through the lists, the algorithm will only project the splats where at least one of the corners has $h > 0$. We do this to avoid projecting the whole dataset for every list (when $h = 0$). For example, in Figure 5, if we are searching through the list of axis X, splats A and C are considered for rendering, and splat B is discarded. If $h = 0$, all splats are projected three times, one time for each list. The value $h = 1$ is not allowed. The closer h is to 1, the less splats are projected more than one time.

The value of w must be interpolated for every pixel of the splat. We use h as one of the parameters for the vertex program. Here, w_i is calculated in hardware, and it is set as one component of the secondary color of the vertex i. Then, the values of w are automatically interpolated by the OpenGL engine. Later, a fragment program stores the interpolated value of w_i as a depth value in the pbuffer.

For each pbuffer, the color and opacity are blended using traditional alpha-blended composition. The weights w stored in the depth buffer do not need to be accumulated, since they are the same for every contribution to the same pixel in the same pbuffer. No depth test is performed.

After processing the splats in all lists, the three textures from each pbuffer are merged using a fragment program. The color C_p for each pixel p is

$$C_p = \frac{\sum_{n=0}^{2} C_n \cdot {w_n}^2}{\sum_{n=0}^{2} {w_n}^2}, \quad (1)$$

where C_n is the color, including the alpha component, and w_n is the weight of the fragment for pbuffer n, and n can be 0 (axis X), 1 (axis Y) or 2 (axis Z). For our implementation, we use $h = 0.85$ (as seen in Figure 6). The higher h is the faster the algorithm runs. However, h cannot be too high, otherwise the effect will be comparable to simply clipping the splats, causing problems similar to those described earlier in this section. For instance, if $h = 0.95$ we are able to see the transition lines in an animation.

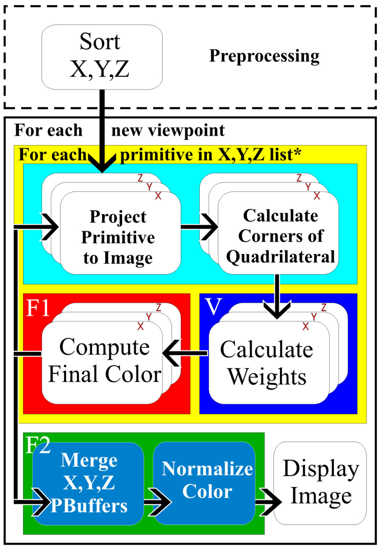


Figure 7. The rendering algorithm with PBuffers. Modules in F1 and F2 are executed by fragment programs. A vertex program executes one module in V.

Figure 7 shows our modified algorithm. Now we cannot clip any quadrilateral, since the areas between the axes regions are smoothed out by the blending of those quadrilaterals. (In our basic algorithm they were clipped against the boundary of the axes regions). Also, notice that now we may search through a list even if no viewing ray has the largest component equal to the one that identifies the list. Instead, we search through any list where $w > 0$ is true for any viewing ray of the image.

6. Results

In this section we show the results of applying our technique to several different types of point-based and volume rendering methods. Unless otherwise noted, all images are rendered using the basic algorithm.

Figure 1 shows the result of our technique applied to classic splatting (Figure 1(a) and (b)) and to the ellipsoid projection method of [16] (Figure 1(c) and (d)) using range data with estimated depth uncertainty [13]. Notice how we cannot see the discontinuity of the axes region borders in (a) and (c).

In (a) and (b) we have used a circular 2D elliptical splat oriented perpendicular to the estimated normal of each point. The model used here is *The Frederick P. Brooks, Jr. Reading Room* in Sitterson Hall at UNC-Chapel Hill. This model has 362880 points. In (c) and (d), instead of only using the depth information per pixel, we compute a depth uncertainty region around it [13]. Next, we render the depth with uncertainty by splatting 3D ellipsoidal Gaussian kernels (based on the EWA Volume Splatting work in [16]). The size of each ellipsoid is given by how uncertain the depth of the point is; the higher the uncertainty, the larger the ellipsoid. Also, the higher the uncertainty, the less weight it has for the final blending. The contribution of the ellipsoid is also weighted according to how close the angle of the current viewpoint is to the original image's viewpoint of this ellipsoid. For more information refer to [13]. The model used here has four 300x300 input images.

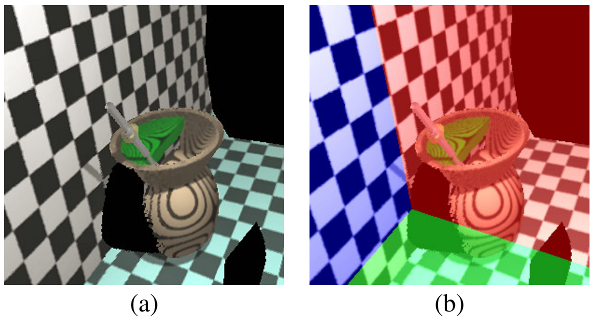


Figure 8. The 'mate' dataset, hyperline rendering. In (b), Red = X axis, Blue = Z axis, Green = Y axis.

Figure 8 shows our technique applied to light field rendering using colored point clouds, also called *hyperline rendering* [17]. Each geometric point is represented in the light field rayspace, and it is called a geometry hyperline. The resulting footprint of each geometry hyperline is blended in a way that it is similar to the splatting technique; therefore, it also needs sorting. The 'mate' dataset is a 3D model obtained from five 300x300 input images of a synthetic model rendered using a ray tracer. Similar to Figure 1, the borders where the projections of axes intersect do not appear in the rendered image.

Figure 9 shows the result of a direct volume rendering algorithm from a foot model. The model is a 183x255x125 rectangular solid with 1-byte of density information for each voxel. It is important to notice that for this kind of model - a structured, regular grid - we do not need to create the sorting lists. Instead, we only have to search through the sheets of every axis, in a front-to-back or back-to-front order (depending on how our blending is performed).

Next, a comparison between using the basic algorithm (with OpenGL clipping planes), and rendering to hardware

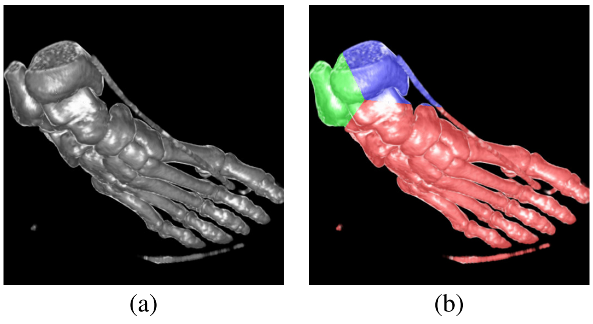


Figure 9. The Foot dataset, 183x255x125 voxels, direct volume rendering. In (b), Red = X axis, Blue = Z axis, Green = Y axis.

Method	Splats	Normal	Optimized
PBuffers h=0.00	2691697	0.790 fps	0.889 fps
PBuffers h=0.85	1559188	1.016 fps	1.307 fps
PBuffers h=0.95	1148354	1.143 fps	1.488 fps
Clipping Planes	956507	1.208 fps	1.600 fps
Axis-Aligned	897229	2.062 fps	2.667 fps

Table 1. Performance comparison for the foot data and viewpoint of Figure 9."Optimized" means optimized for memory access.

pbuffers which are later blended, is shown in Figure 10. Figure 10b shows how the artifacts usually appear when using the basic algorithm in volumetric data. Since the artifacts are best viewed in computer animations we refer readers to **http://www.inf.unisinos.br/∼chris/sibgrapi2005** .

Finally, figure 11 shows the foot dataset with a different opacity and color map, with both inner (bone) and outer (flesh) structures.

The performance comparison between the basic algorithm with OpenGL clipping planes, the rendering to pbuffers with different values of h, and Westover's axis-aligned implementation are shown in Table 1 (for a viewpoint where the all three axes lists are projected) and Table 2 (for an image plane parallel to the $X-Y$ plane, where only the list for axis Z is projected). The resolution of the rendered images is 512 x 512 pixels. The number of splats indicates how many splats were projected for each method. The optimized version of our algorithms replicates the data for each list, without including points where opacity is equal to zero. In this case, if the opacity map changes, the lists have to be recreated. The foot model has 5833125 points, of which 897229 have opacity greater than zero for the opacity map used in our tests. Our algorithms were implemented in OpenGL using Windows XP running on a 3.0 Ghz Pentium

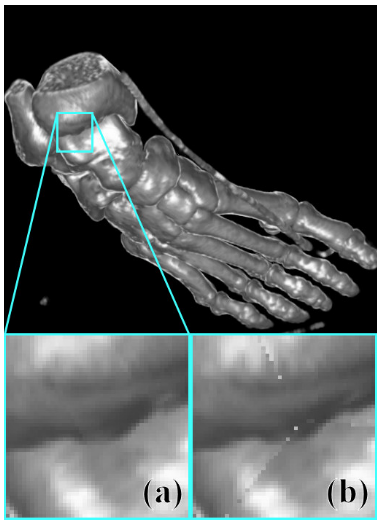


Figure 10. Top: foot dataset rendered with three pbuffers blended using formula 1; bottom: comparison between close-up of this image (a) with the same area from the image rendered with OpenGL clipping planes (b) (from Figure 9).

4 with 2 Gb memory, and an NVIDIA 6600 GT graphics card with 128 Mb memory. Notice that with the exception of the methods where $h = 0$, which we do not advocate, all methods achieved frame rates above 1 frame per second.

7. Conclusions

We have presented here a new, hardware-assisted technique for rendering splats while maintaining the correct visibility ordering. Our basic algorithm is simple and it is easily applied to any point-based algorithm that needs viewpoint sorting for rendering. Also, the render to pbuffer version our algorithm is especially interesting for applications that need more visual information than just the implicit sur-

Method	Splats	Normal	Optimized
PBuffers h=0.00	897229	0.970 fps	1.185 fps
PBuffers h=0.85	897229	1.333 fps	1.730 fps
PBuffers h=0.95	897229	1.333 fps	1.730 fps
Clipping Planes	897229	1.333 fps	1.730 fps
Axis-Aligned	897229	2.208 fps	2.667 fps

Table 2. Performance comparison for the foot data and a lateral viewpoint, where only one axis list is selected.

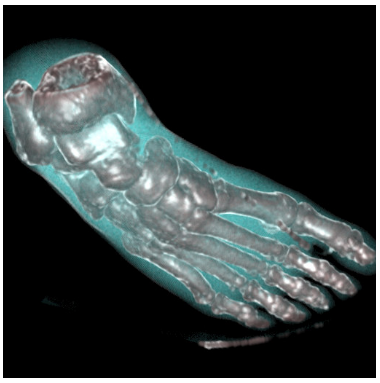


Figure 11. Foot dataset showing flesh and bone, rendered with three merged pbuffers.

face of the objects, as is usually the case for volume rendering, and volume splatting from depth with uncertainty.

In the future we plan to further exploit our technique using the capabilities of the newest graphics hardware in the market. As an example, it is straightforward to apply our technique to the real-time implementation in [12].

Acknowledgments

The authors would like to thank Lars Nielsen, Anselmo Lastra, Nick England and David McAllister of the University of Carolina at Chapel Hill for the reading room model; George Chen from STMicroelectronics for the camera calibration of the dataset in Figure 1c; Marcelo Walter and Berenice F. Hofsetz for proofreading the first draft of the paper; Mauro Steigleder for helping with the pbuffer code; and Rui Bastos from NVIDIA for providing the GeForce 6600 GT.

References

[1] Levoy, M. and Whitted, T., *The Use of Points as a Display Primitive*, Technical Report TR 85-022, Univ. of North Carolina at Chapel Hill (1985).

[2] Zwicker, M., Pfister, H., van Baar, J. and Gross, M., *Surface Splatting*, Proc. of SIGGRAPH 2001 (2001), pp. 371–378.

[3] Westover, L., *Interactive Volume Rendering*, Proc. of the Chapel Hill workshop on Volume visualization (1989), pp. 9–16.

[4] Grossman, J. P. and Dally, W. J., *Point Sample Rendering*, Proc. of 9th Eurographics Workshop on Rendering (1998), pp. 181–192.

[5] Laur, D., Hanrahan, P., *Hierarchical splatting: a progressive refinement algorithm for volume rendering*, Proc. of SIGGRAPH 91 (1991), pp. 285–288.

[6] Shade, J. W., Gortler, S. J. and Szeliski, R., *Layered Depth Images*, Proc. of SIGGRAPH 98 (1998), pp. 231–242.

[7] Gortler, S. J. and He, L-w, *Rendering Layered Depth Images*, Technical Report MSTR-TR-97-09, Microsoft Research (1997).

[8] Max, N., *Hierarchical Rendering of Trees from Precomputed multi-layer Z-buffers*, Rendering Techniques '96 (1996), Vienna: Springer-Verlag, pp. 165–174.

[9] Pfister, H., Zwicker, M., Baar, J. v. and Gross, M., *Surfels: Surface Elements as Rendering Primitives*, Proc. of SIGGRAPH 2000 (2000), pp. 335–342.

[10] Liu, R., Pfister, H., Zwicker, M., *Object Space EWA Surface Splatting: A Hardware Accelerated Approach to High Quality Point Rendering*, EUROGRAPHICS 2002, Computer Graphics Forum, Vol. 21, No. 3, 2002, pp.461-470.

[11] Mueller, K., Yagel, R., *Eliminating Popping Artifacts in Sheet Buffer-Based Splatting*, Proceedings of IEEE Conference on Visualization 1996, pp.65-72, 1996.

[12] Chen, W., Ren, L., Zwicker, M., Pfister, H., *Hardware-Accelerated Adaptive EWA Volume Splatting*, Proceedings of the IEEE Visualization 2004, pp. 67 - 74.

[13] Hofsetz, C., Ng, K. C., Max, N., Chen, G., Liu, Y., McGuinness, P., *Image-Based Rendering of Range Data with Estimated Depth Uncertainty*, IEEE Computer Graphics and Applications, Vol. 24, No. 4, July 2004, pp.34-42.

[14] Räsänen, J. *Surface Splatting: Theory, Extensions and Implementation*, Master's Thesis, Helsinki University of Technology (2002), pp. 19-44.

[15] Hofsetz, C. *Image-Based Rendering of Range Data with Depth Uncertainty*, Doctoral Dissertation, University of California, Davis, 2003.

[16] Zwicker, M., Pfister, H., van Baar, J., Gross, M., *EWA Volume Splatting*, IEEE Transactions on Visualization and Computer Graphics,Vol. 8, No. 3, July-Sept. 2002, pp. 223-238.

[17] Hofsetz, C., Chen, G., Max, N., Ng, K. C., Liu, Y., Hong, L., McGuinness, P., *Light Field Rendering Using Colored Point Clouds - A Dual Space Approach*, Presence: Teleoperators and Virtual Environments, Vol. 13, No. 6, MIT Press, 2004, pp.726-741.