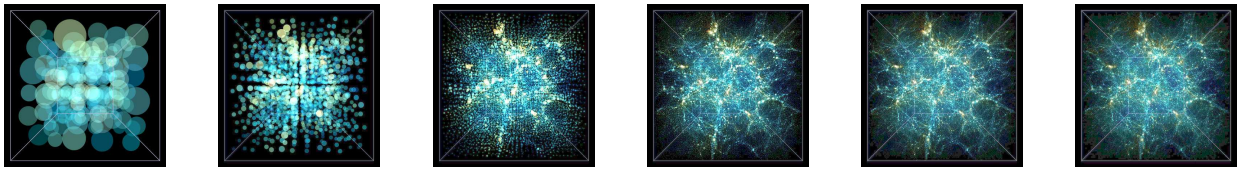
Hierarchical Splatting of Scattered Data

Matthias Hopf* Thomas Ertl*
Visualization and Interactive Systems Group
University of Stuttgart, Germany



Abstract

Numerical particle simulations and astronomical observations create huge data sets containing uncorrelated 3D points of varying size. These data sets cannot be visualized interactively by simply rendering millions of colored points for each frame. Therefore, in many visualization applications a scalar density corresponding to the point distribution is resampled on a regular grid for direct volume rendering. However, many fine details are usually lost for voxel resolutions which still allow interactive visualization on standard workstations. Since no surface geometry is associated with our data sets, the recently introduced point-based rendering algorithms cannot be applied as well.

In this paper we propose to accelerate the visualization of scattered point data by a hierarchical data structure based on a PCA clustering procedure. By traversing this structure for each frame we can trade-off rendering speed vs. image quality. Our scheme also reduces memory consumption by using quantized relative coordinates and it allows for fast sorting of semi-transparent clusters. We analyze various software and hardware implementations of our renderer and demonstrate that we can now visualize data sets with tens of millions of points interactively with sub-pixel screen space error on current PC graphics hardware employing advanced vertex shader functionality.

Keywords: Volume Rendering, Scattered Data, Splatting, Hierarchical Visualization

1 Introduction

Quite a number of physical simulations create large point-based data sets, for example Smoothed Particle Hydrodynamics (SPH) [Monaghan 1992] and n-body simulations [Jenkins et al. 1998] in astrophysics. Other sources of scattered point data are astronomical observations where new techniques for measuring three dimensional positions of stars as in the GAIA project [GAIA 2003] will create huge real-world data sets in the near future as well. These data sets contain up to hundreds of millions of points each with information about position $\mathbf{x}_i$, diameter s_i, and intensity c_i at various wavelengths.

*{hopf,ertl}@vis.uni-stuttgart.de

These data sets are too large to be rendered in their entirety at interactive frame rates and the memory requirements are quite problematic for standard PCs as well. An alternative approach [Kähler et al. 2002; Park et al. 2002] is to resample the data sets and use standard volume visualization, which can be implemented quite efficiently using texture mapping [Rezk-Salama et al. 2000]. However, this technique imposes a low-pass filter on the data set, and for reasonable frame rates and memory usage the filter domain is so large that almost no subtle details within the data will be visible any more. To some extent this can be avoided for off-line rendering of animations. In this case hierarchical volume scene graphs can be used e.g. for visualizing stellar nebula [Nadeau et al. 2000].

In order to allow scientists to view these data sets at high resolution interactively on desktop workstations or PCs, we want to visualize the scattered data directly without resampling them to a volume density. We can achieve significant speedup by applying clustering techniques to create a hierarchical representation of the data set. The hierarchy can then be rendered adaptively according to screen resolution and focus points, and a lower hierarchy level can be chosen for the visualization during interaction. Of course, hierarchical data structures generate additional memory overhead imposing even greater restrictions on the maximum data size, but storage requirements can be reduced using relative position coding, while still maintaining high accuracy with respect to the particle positions.

In order to visualize scattered data interactively the point coordinates have to be transformed into image space and rasterized into the frame buffer. Current graphics hardware is highly optimized for this task and frees up the CPU for concurrent hierarchy selection and traversal. As triangles are the dominating primitive in computer games, rasterization throughput may be higher for polygons than for points. However, this will have no major effect, since our approach is more likely to be geometry limited rather than rasterization limited, because large numbers of points can only be visually precepted well as long as they do not overlap too much. For certain types of data — e.g. with widely varying point sizes or semitransparent appearance — blending may be necessary in order to enable visual depth perception. This requires the points to be sorted according to their projected z coordinates. Due to the high number of points this is nontrivial to do in realtime, but can be efficiently implemented based on our hierarchical data structures.

2 Previous Work

There has been quite a lot of work in the area of using footprints as rendering primitives for sampled data. Laur and Hanrahan [1991] introduced hierarchical splatting for volume rendering using Gouraud-shaded polygons. Researchers like Mueller et al. [1999], Swan et al. [1997], and Zwicker et al. [2001a] focus mainly on the improvement of the visual quality of texture splatting; however, the techniques described in these papers only apply

to the reconstruction of continuous functions e.g. for volume rendering of regular grid data, and they do not address adaptive rendering or data size reduction. Additionally, there exist a number of non-realtime rendering systems for large point-based data sets, e.g. for rendering film sequences [Cox 1996].

Using points as rendering primitives is a topic of ongoing research. However, almost all publications in this area deal with the rendering of geometric surfaces. Alexa et al. [2001], Pfister et al. [2000], Rusinkiewicz and Levoy [2000], Wand et al. [2001], and Zwicker et al. [2001b] showed different methods to create data hierarchies of surfaces represented by sample points and how to render them efficiently. As the intrinsic model of points describing a surface is fundamentally different to the model used for scattered data, their clustering techniques cannot be applied in our case. Pauly et al. [2002] used principal component analysis for clustering, but with a different hierarchy concept compared to our approach. Some systems [Rusinkiewicz and Levoy 2000; Botsch et al. 2002] use quantized relative coordinates for storing the points in a hierarchical data structure, but these approaches were not optimized for fast GPU access because the data structures had to be interpreted by the CPU. Additionally, the presented rendering techniques have been designed to create smooth surfaces without holes and they allow no or only few layers of transparency. Again, this does not meet our requirements.

First steps for visualizing uncorrelated samples for SPH data have been presented by Rau and Straßer [1995]. Jang et al. [2002] introduced a multiresolution splatting approach for non-uniform data. However, in their solution the higher hierarchy levels are always stored in uniform grids, and they cannot render more than approximately 135,000 splats per second. This technique seems to be more appropriate for almost flat and regular data.

For rendering large quantities a simple brute force approach would store the complete data set on the graphics card and use point array rendering for displaying the data set. As soon as the data set does not fit into graphics memory, rendering speed can drop by an order of magnitude.

In the following we propose a hierarchical data structure based on principal component analysis or similar clustering techniques that enables us to render large data sets adaptively at high frame rates on current PC hardware without compromising visual quality. Our scheme also reduces memory consumption by using quantized relative coordinates and it allows for fast sorting of semi-transparent clusters.

3 Data Storage

Using hierarchical structures imposes higher memory requirements than storing the same data in flat arrays. A trivial implementation can easily exhaust main memory on regular workstations for large data sets, even in the steady state. Memory bandwidth is limited, and traversing the hierarchy for rendering adds overhead for recursive function calls and pointer dereferencing. Additionally, with current graphics APIs there is no means to hand this process over to the GPU.

Therefore, we decoupled hierarchy structures (clusters) from data structures (points). The clusters contain a pointer to the next hierarchy level, a pointer to offspring point data, and the number of children. The point data itself only contains the point position, size, and color values. In principle one would like to store raw data values and use runtime classification for point size and color selection, but the cluster hierarchy itself and especially the pre-processed cluster representatives highly depend on point sizes and colors.

Figure 1 shows the lowest three levels of a typical data hierarchy. The finest level n does not contain any hierarchy information at all, thus no cluster nodes are needed. In level $n-1$ a point data structure is related to each cluster node, containing its centroid. Cluster nodes and point data are connected on the previous level. The diagram shows which points are rendered for a typical cluster node for rendering levels $n-2$, $n-1$, or n.

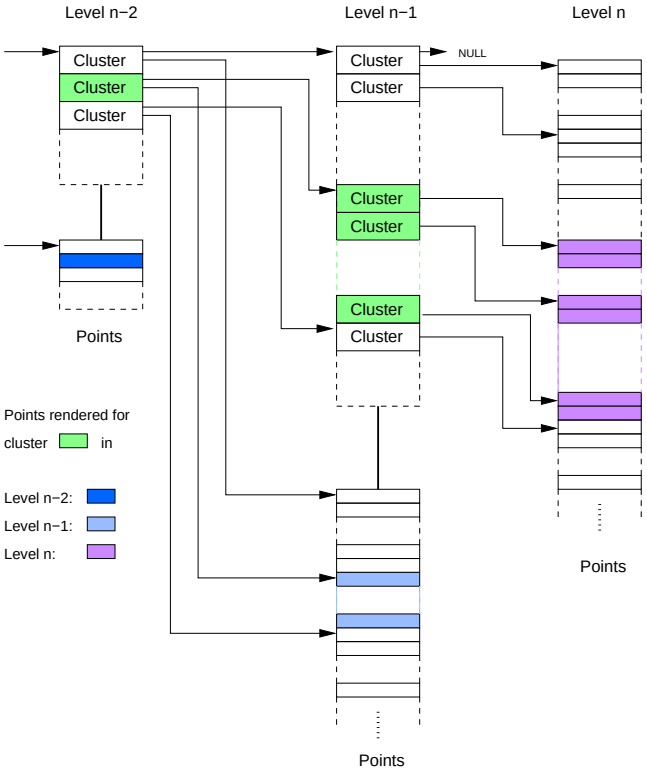


Figure 1: The last three levels of a typical data hierarchy. The green clusters can be rendered at their highest hierarchy level in a single loop without recursively descending the data structure. The point data correlated to the cluster centroids is not embedded in the clusters but stored in point structures parallel to the clusters.

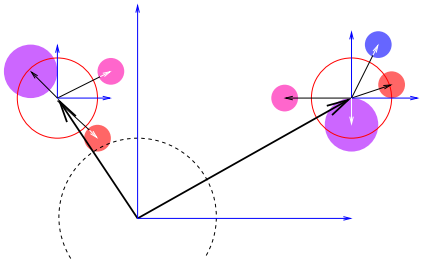


Figure 2: Point coordinates are scaled and quantized relatively to the position of the cluster centroid for storage.

The decoupled data structure enables us to store point data for any given combination of rendering level and cluster node in a continuous array. This reduces the number of necessary recursive function calls and helps us with accelerating rendering by using graphics hardware. Additionally, it ensures that the data is concatenated for efficient cache usage.

Point data sets tend to get really large, and they need high positional resolution. Memory requirements can be reduced to one half or even one quarter by storing coordinates relatively to the centroids of the inspected clusters as depicted in Fig. 2. As the necessary positional resolution is much lower for encoding relative coordinates, they can be quantized using bytes or shorts instead of floats.

Finally, for a typical data set like the VIRGO n-body simulation (Fig. 8) with 16 million points in level 6 we need 160 Mbyte for point data and 32 Mbyte for the cluster hierarchy when storing point coordinates in bytes only.

- Select cluster j (point indices I_j) with largest distortion Δ_j
- Calculate auto-covariance matrix from centroid X_j: $A = \sum_{i \in I_j} (\mathbf{x}_i - \mathbf{X}_j)(\mathbf{x}_i - \mathbf{X}_j)^T$
- Find Eigenvector $e_{\max}$ of A corresponding to the largest Eigenvalue $\lambda_{\max}$
- Split cluster j into two new clusters: $I_{n1} = \{i \in I_j : \langle \mathbf{x}_i - \mathbf{X}_j, e_{\max} \rangle \geq 0\}$ $I_{n2} = \{i \in I_j : \langle \mathbf{x}_i - \mathbf{X}_j, e_{\max} \rangle < 0\}$
- Calculate centroids and distortions for the new clusters

Figure 3: The PCA-split algorithm

4 Creating the Hierarchy

In order to create one level of the hierarchy the input data points have to be sorted into bins. For each bin a point on the next higher hierarchy level is created, representing all points that fell into that bin. The properties of the newly created point are chosen so that its visual representation matches that of the substituted points best.

To obtain the set of bins several clustering schemes can be used. The most common solution is to subdivide the data set into an octree, which can be used efficiently for sorting as well (Sect. 6).

Another approach that has much better spatial adaptation properties is to perform a series of principal component analysis (PCA) splits, each dividing the cluster with the currently highest distortion as defined below into two halves. As PCA is a standard technique, we only present a short summary of the PCA-split algorithm in Fig. 3, more details can be found e.g. in [Jolliffe 1986].

In the following I_j denotes the set of indices of the points of cluster j. That is, cluster j consists of all points $\mathbf{x}_i$, $i \in I_j$ with diameters s_i, and has the weighted centroid $\mathbf{X}_j$ and distortion Δ_j with

$$\mathbf{X}_j = \frac{\sum_{i \in I_j} s_i \cdot \mathbf{x}_i}{\sum_{i \in I_j} s_i} \quad , \qquad \Delta_j^2 = \sum_{i \in I_j} \left\| \mathbf{x}_i - \mathbf{X}_j \right\|_2^2 \quad .$$

As we have to perform this split operation several million times, fast cluster selection is of uttermost importance. Therefore, we keep the clusters in a skip-list [Pugh 1990], sorted by decreasing Δ_j^2. A skip-list is essentially a sorted linked list with randomized link depth, with $O(\log n)$ complexity in the average case for search, insert, and delete operations. Its properties are similar to balanced trees, with the advantage of faster insert and delete, $O(1)$ largest value search, very small memory footprint, and almost trivial implementation.

This splitting process is continued until the maximum distortion or the average cluster size fall below pre-defined minima. After the visual properties of the new points have been obtained, these points undergo another series of PCA-splits in order to create the next hierarchy level. For most applications like the VIRGO data set a hierarchy depth of more than about 6 levels is usually not appropriate. For this data set with its 16 million points the hierarchy creation process takes only a few minutes.

For creating a visually approximative representation of the cluster j compared to its children the most important aspect is that the radiant flux Φ has to be the same. For the irradiance c_j of the new centroid point representing the cluster this means for each of the representative wave lengths

$$\Phi_j = A_j \cdot c_j = \frac{\pi}{4} s_j^2 \, c_j = \frac{\pi}{4} \sum_{i \in I_j} s_i^2 \, c_i \quad . \tag{1}$$

The cluster representative should be larger than the largest of its children in order to keep some visual continuity. Additionally, small cluster points would have very high local intensities, which could finally saturate the covered pixels in the blending step during rendering. Distributed clusters — that is clusters with a large average distance of their children to the centroid compared to the children's point sizes — should have larger representatives than locally agglomerated ones. On the other hand, they must not be too large, as the human eye is very sensitive to edges, and enlarging a point implies reducing its intensity, diminishing the visibility of the edge.

After comparing several different functions, we found a trade-off that creates acceptable results for almost all point distributions. It tries to combine point sizes and their distances to the centroid, and ensures, that the final size does not fall below the size of the largest point of the cluster:

$$\begin{aligned} m_j &= \operatorname*{argmax}_{i \in I_j} s_i \quad , \\ s_j &= \frac{0.5}{|I_j| - 1} \sqrt{\sum_{i \in I_j \setminus \{m_j\}} s_i \left\| \mathbf{x}_i - \mathbf{X}_j \right\|_2} \; + \; s_{m_j} \quad . \end{aligned} \tag{2}$$

The scaling factor $\frac{1}{2}$ in Eq. 2 of the weighted average point size of all points except the largest one before adding to the largest point size s_{m_j} has been determined empirically.

This calculated point size is subject to further restrictions, if intensities are stored in main memory as unsigned bytes in order to save memory. The system has to assure that the calculated point size does not overflow the intensity domain, and it has to increase the point size in case of saturation.

Eq. 1 and Eq. 2 are highly dependent on the blending function, and our definition only holds for cumulative blending ($C = c_1 + c_2$). For other blending functions, like the over operator ($C = \alpha_1 c_1 + (1 - \alpha_1) c_2$), c_j may be view dependent, as the total flux of overlapping points is no longer necessarily the sum of the individual fluxes of the points. With the current approach view dependent intensities cannot be modeled. However, with adaptive rendering we will use coarser hierarchy levels only for clusters that are projected to areas on the screen that are small or outside some region of interest, and it is very unlikely that view dependencies will be noticed in such small regions.

5 Hierarchy Traversal

During rendering the hierarchy is traversed recursively. For each cluster the system may decide to descend further down into the hierarchy, render the centroid at the current level, or skip the cluster altogether when it is not visible. The decision can be based upon some maximum screen error metrics, the distance to the viewer, or some given region of interest. These rules should be computationally cheap. As a rule of thumb, evaluating the rule should be cheaper than transforming and rendering one point of the cluster.

For more complex rules and for accelerating the traversal process, the system may already decide on a higher level n, that it will render all offsprings of level $n + \delta_n$ (see Fig. 4). Then the children do not have to be traversed. Even for simple adaptivity rules this has a strong effect on rendering performance. As described in Sect. 3, the point data of all children is stored linearly in memory, thus they can be addressed in a single loop, or even by a single OpenGL array rendering call. See Fig. 5 for a pseudo code fragment.

Remember that we are using relative coordinates for storing the point locations. In this context children of different clusters can only be rendered in a single loop when the base centroid and the scaling factor for the relative coordinates is the same for all considered children. We can use the coordinates of a centroid of level n for the calculation of the relative coordinates of all descendants of level $n + \delta_n$. In order to use this arrangement efficiently, δ_n has to be constant for the data set. We get a speedup of about 50 percent for an average cluster size of 5 points and $\delta_n = 2$, more for larger clusters. For adaptive rendering higher δ_n are less efficient as the traversal routine has to select the clusters to be rendered on a higher level. Another drawback is that being able to render a set of clusters in one piece comes at the cost of higher discretization errors.

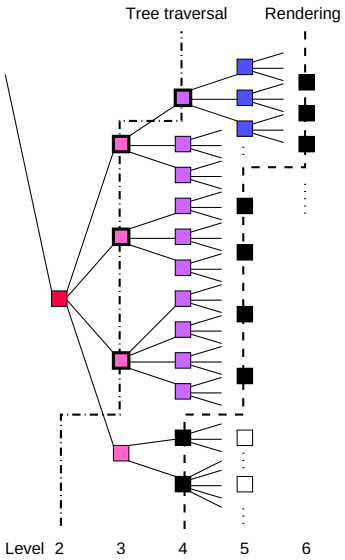


Figure 4: During traversal, the final rendering level should be selected at some higher level of the hierarchy for speedup reasons ($\delta_n = 2$ in this case).

```
void render_cluster (cluster_t *c, point_t *p) {
  if (cluster_visible (c)) {                    /* trivial reject */
    if (descend_cluster (c, p, δn)) {
      for i = 0...c→len[0]                      /* recursion */
        render_cluster (&c→children[i], &c→points[i])
    } else {
      len = c→len[δn−1]
      for i = 0...δn−2          /* find first point of hierarchy depth δn */
        c = c→children[0]              /* not executed for δn = 1 */
      render_points (c→points, len)
} } }
```

Figure 5: Pseudo code for traversing the hierarchy.

6 Sorting

For many of the investigated data types cumulative blending is an effective way of visualizing both global and local structures in the data sets. However, with other data sets, for instance reversible Apollonian packings (Fig. 20), the over operator is necessary to visualize the visual depth. Non-commutative blending operators require the data points to be sorted according to view distance. The implemented hierarchy can be used to efficiently sort the cluster centroids using quicksort or bucketsort. The cluster points themselves have to be sorted before rendering as well. Bucketsort only creates an approximative sorting order, but has the advantage of lower computational complexity ($O(n)$ vs. $O(n \log n)$) and its implementation is much simpler and thus faster. The rendered images are almost indistinguishable when using a relatively large number of buckets.

Sorting the cluster centroids is equivalent to the typical BSP-tree sorting, as long as the distances of any two neighboring centroids to their common splitting plane are equal (Fig. 6). The octree clustering approach has this property, however, its spatial adaption to the local point density is much worse than the proposed PCA-split approach. Still, we have no choice but to use octrees if we really care about the correct sorting order.

Even with correctly sorted clusters, there is a chance that overlapping points are rendered in the wrong order, as it can be seen in Fig. 7. For many data sets the points can be thought to be infinitely small, in that case the points are rendered correctly. Other data sets are more sensitive to their sorting order, and require larger average cluster sizes by combining several octree levels to a single level. This helps reducing the chance of sorting errors, as the points of a single cluster are always rendered in the correct order.

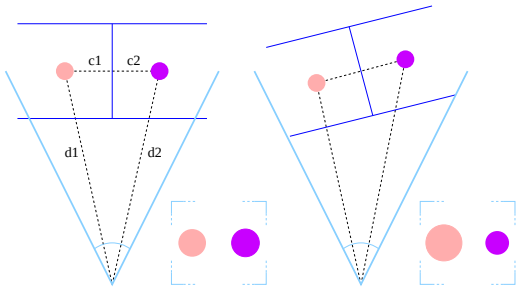


Figure 6: Distance sorting according to $d1$, $d2$ is equivalent to BSP sorting for $c1 = c2$. Note that the sorting order of $d1$, $d2$ changes exactly at the same time the visibility order of the two cells changes.

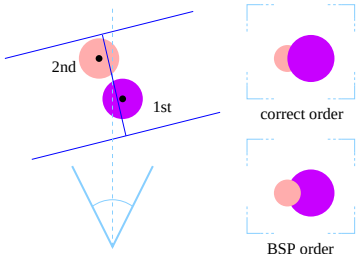


Figure 7: Back-to-front distance sorting according to BSP fails for non-split overlapping points. In this example the right cluster is rendered before the left one due to BSP order.

In order to sort the Voronoi cells produced by the PCA-splits, additional connectivity and splitting plane information is needed for MPVO [Max et al. 1990; Williams 1992] or equivalent algorithms. This implies a huge additional memory overhead we would like to avoid. It is ongoing research, how this approach can be combined with per-pixel clipping or z-test dependent blending to render exactly sorted images even for cases like in Fig. 7.

7 Rasterization

Since using only one vertex per primitive can accelerate the rendering process significantly, we will usually approximate the splats using OpenGL anti-aliased points. Other footprints can be used by rendering point sprites without additional cost (see Fig. 20), but they are only available on NVidia hardware right now. For rendering large quantities of points the generally fastest approach is to use vertex coordinates and attributes that are given by vertex arrays or display lists. However, display lists have to be stored in precious graphics memory and are more likely to be larger in size, as the graphics card has to store additional information about its contents and format.

When a point projects to an area with diameter $\tilde{s}$ smaller than a single pixel on the screen, its brightness has to be attenuated. The new alpha value is

$$\tilde{\alpha} = \alpha \cdot \tilde{s}^2 \quad , \tag{3}$$

assuming that point color is multiplied with alpha during blending. Note that attenuation increases quantization artifacts due to the limited frame buffer depth. Therefore, adaptive rendering can even improve the image quality by choosing higher levels for parts of the cluster that tend to project to very small screen areas.

For drawing points with varying sizes we can use vertex programs, a concept NVidia introduced with the GeForce3 (`NV_vertex_program`). Besides changing the point size on a per vertex level and adding the last contribution of the relative coordinates, the vertex shader is responsible for correct alpha attenuation as indicated in Eq. 3, which is not possible without using vertex programs at all when we want to employ vertex arrays. Figure 15

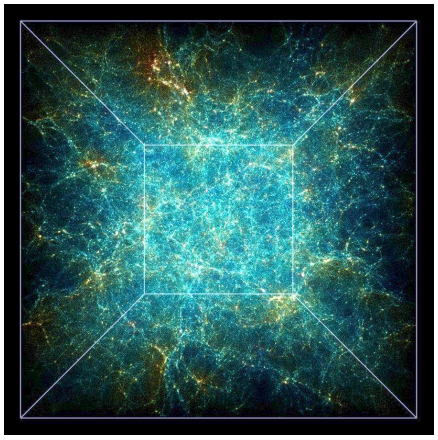


Figure 8: A total view of one of the VIRGO n-body simulations with 16.8 million particles (level 6).

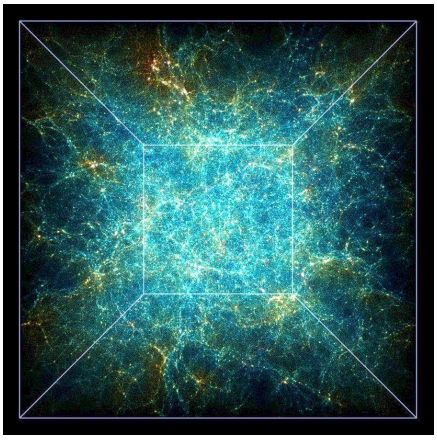


Figure 9: A total view of one of the VIRGO n-body simulations rendered adaptively with a maximum screen space error of 2 pixels.

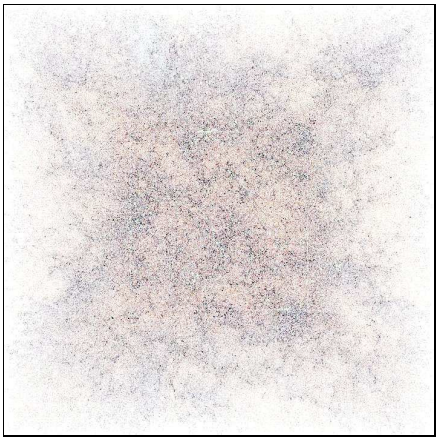


Figure 10: Differences of adaptive vs. full data rendering (contrast enhanced by 400%).

shows the program parameter configuration and the actual vertex program written in Cg [Mark et al. 2003] that contains all of the above. Additionally, point size and alpha values are multiplied by two global scaling factors. It compiles into 29 program statements for both `NV_vertex_program` and `ARB_vertex_program`.

The later extension, which is supported by ATI's Radeon 9700, finally enables us to evaluate the algorithm on this card as well. The previous `EXT_vertex_shader` extension did not allow us to alter the point size on a per-vertex basis. As most of the performance gain comes from this last step, we could not really benefit from the Radeon's high performance geometry engine with the old extension. Unfortunately, both anti-aliased point image quality and execution speed is clearly below our expectations with the current drivers (see Sect. 9 for a comparison). Note that at least the setup is still much slower on NVidia cards as well compared to `NV_vertex_program`.

ATI's DirectX drivers are more mature than their OpenGL drivers, thus we investigated whether this API would be an option. However, the so-called flexible vertex format of DirectX up to version 8 only supports vertices specified as floats. As we do not want to store the points' vertices in this format due to its memory requirements, we would have to convert them on the fly, which would make the use of vertex buffers extremely expensive as they would have to be converted by the CPU.

With the availability of vertex shaders we can now use vertex arrays to send a large part of the hierarchy to the graphics hardware. When sorting is enabled, index arrays have to be used to select the points in the correct order. These calls are highly optimized, and the CPU can already continue to select the next cluster to be rendered in parallel to the rasterization process. As pointed out in Sect. 5 we have to take care that we only send down that part of the hierarchy in one piece that is related to the same base centroid for the calculation of the relative coordinates.

8 Alternative Rendering Approaches

For comparison, several other techniques have been developed and integrated into the rendering framework. The different rendering backends can be selected during runtime at almost no cost.

As hierarchy traversal and coordinate transformation seem to be the limiting factors for the visualization of scattered data, a software rasterizer is a valid option to be considered. Most points of a low hierarchy level project to a very small area on the screen, so the rasterizer should be optimized for single pixel points. This implementation can also function as a reference for the OpenGL based render backends, as it draws the points to a floating point frame buffer. With this feature we reduce the chance of missing contributions of very small or dim points. However, the CPU is completely responsible for vertex transformation and rasterization, thus this solution is most likely to be the least efficient of the presented methods, and new graphics hardware like the Radeon 9700 or the GeForce FX is able to render into floating point frame buffers as well.

In contrast to regular PC workstations used by typical end users, virtual reality environments are still often based on Silicon Graphics systems. As the InfiniteReality hardware does not have a programmable graphics pipe, we additionally implemented a regular billboard renderer. Using billboards is less efficient compared to OpenGL anti-aliased points or point sprites, as four vertices have to be calculated and sent down the pipeline for a single data point.

Note that rendering points without vertex programs is not an option, as with the regular OpenGL pipeline one can only set the current point size outside an `glBegin()` / `glEnd()` pair, which reduces the overall speed considerably due to the state changes.

9 Results and Discussion

The images 8, 9, and 17 show visualizations of n-body simulations carried out by the Virgo Supercomputing Consortium. All images show redshift $z = 0$ for the τCDM model. The velocities of the galaxies relative to the simulated base cube have been color coded.

In images 11 to 13 one can see different levels of the data set. Note that level 3 would usually not be used for rendering, but it is a potential level for deciding on the rendering depth, as shown in Fig. 4. Figures 18 to 20 show other data sets and rendering modes. Please note that the clearly visible aliasing in Fig. 18 is inherent to the according data set and not an artefact of the presented rendering technique. In most areas the data set contains an almost regular grid and the splats are used for visualizing the grid structure and not for approximating any underlying function.

Despite the speed of modern processors, the OpenGL accelerated version is still superior to the software approach, which employed a very crude rasterizer in our implementation that renders large points in poor quality only. One major drawback of the software-based system is that the floating point frame buffer has to be sent down the AGP bus to the graphics card, though with latest AGP 8x graphics hardware and current drivers this only imposes an upper limit of 40 fps for a 1000^2 viewport, not including the time for clearing and rendering the software buffer. However, software based rendering still seems to be one of the slowest approaches. Using a 24 bit frame buffer could accelerate this process, but then we loose the major advantage of the software solution.

Level	# Points	Av. pt. size	Software	Billboards	Vertexprogs	V.p.a. $\delta_n = 1$	V.p.a.	V.p.a. adapt. $\delta_n = 1$	V.p.a. adapt.	ATI V.p.a.†
6	16.8M	0.5	847	1724	495	1389	427	104	153	1490
5	3.3M	0.6	433	752	229	262	85	93	79	287
4	671K	0.9	120	161	44	50	17	17	47	57
3	123K	1.5	48	30	8.8	10	3.7	3.7	10	12
2	24K	2.9	26	7.6	2.6	2.7	1.8	1.8	2.7	2.6

† `ARB_vertex_program` with vertex arrays, evaluated on ATI's Radeon 9700, WindowsXP, $\delta_n = 2$

V.p.a. Vertex program with array rendering

Table 1: Rendering times in ms for different rendering techniques and levels for a 500^2 viewport, $\delta_n = 2$ except where noted.

Level	# Points	Soft* $\delta_n = 1$	Soft† $\delta_n = 1$	Soft†	Soft‡	V.p.a.* $\delta_n = 1$	V.p.a.† $\delta_n = 1$	V.p.a.†	V.p.a.‡	V.p.a. adapt.‡ $\delta_n = 1$	V.p.a. adapt.‡
6	16.8M	6670	5880	4760	3570	5880	5260	3125	1890	252	690
5	3.3M	1320	1250	1040	752	1100	1040	658	379	233	356
4	671K	298	282	238	182	220	204	134	78	126	78
3	123K	85	81	71	61	44	40	26	16	28	16
2	24K	41	40	28	29	10	8.2	3.2	3.2	6.1	3.2

Soft Software rendering

V.p.a. Vertex program with array rendering

* quicksort

† bucketsort, # of buckets $= \max(16 \cdot \#\text{ of points per cluster}, 1024)$

‡ bucketsort, # of buckets $= \max(\#\text{ of points per cluster}, 128)$

Table 2: Rendering times in ms for rendering sorted points with correct blending for a 500^2 viewport, $\delta_n = 2$ except where noted.

Viewport	Rendering time	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6
160^2	9.5	0	4K	51K	1.7K	0	0
400^2	75	0	2	53K	348K	224K	23K
700^2	348	0	0	418	343K	1.4M	1.8M
1000^2	769	0	0	0	82K	1.9M	6.2M

Table 3: Number of rendered points per level and rendering times in ms for adaptive rendering vs. viewport size, $\delta_n = 1$.

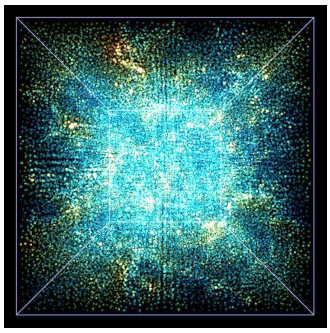


Figure 11: VIRGO at level 3. (123,000 clusters = 0.74%) Rendered at 270 fps.

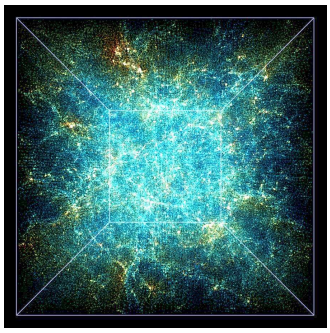


Figure 12: Level 4. (671,000 clusters = 4%) Rendered at 59 fps.

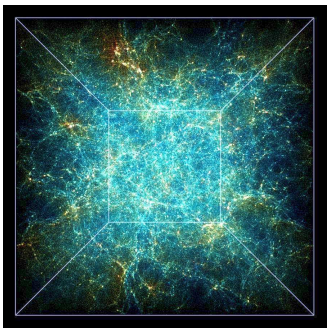


Figure 13: Level 5. (3.3 million clusters = 19.7%) Rendered at 12 fps.

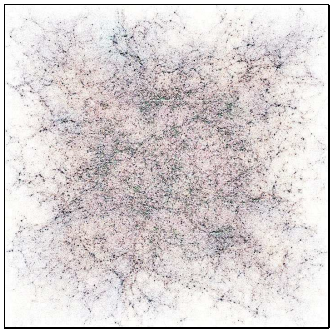


Figure 14: Differences of software vs. OpenGL rasterization (contrast enhanced by 400%).

Table 1 lists some performance measurements for the different algorithms and levels for $\delta_n = 2$, except where noted, together with the number of points, and the average projected size. It can be noticed that using billboards is rather slow, as the CPU has additional work to do for setting up four times the amount of vertices to be sent to the graphics pipe. The system used for the evaluation was an Pentium4 2800 MHz with an Intel 7225 chipset with 4 GB dual channel DDR 333 memory and a GeForce FX 5800 Ultra graphics pipe on a Linux system, except where noted. The Windows XP drivers showed similar but slightly lower performance figures for the GeForce. The Linux drivers for the Radeon currently don't have support for the used chipset. Please note that the high performance memory setup has a much larger impact on the software rasterizer than on the vertex array renderer.

The adaptive algorithm shown in the table uses a simple adaptive scheme with vertex programs and vertex arrays, selecting the clusters that should be traversed on the CPU by the maximum screen projection size of the cluster children and the given maximum traversal depth. If the projected size exceeds 2 pixels, the cluster is traversed further, otherwise its children are rendered to screen for $\delta_n = 1$, for $\delta_n = 2$ the same criterion is applied to its grandchildren. Using these settings, there is almost no visual difference between the data set rendered in full resolution compared to the adaptive rendering. With $\delta_n = 1$ we get a finer hierarchy selection, but we also reduce the average array size that can be used for rendering, which explains the performance loss in the lower levels. The difference image in Fig. 10 shows quite some changes in the visualization, however, they have the same quality as additional noise and do not disturb the visual appearance. Most of the screen space difference comes from points that happen to be rendered one pixel off to their original positions. While the human visual system is not able to notice these differences, they have a rather large impact on difference images. We also noticed that the quantization and floating point roundoff errors introduced by using graphics hardware for rendering (Fig. 14) are larger than the ones created by adaptive rendering. The contrast of both difference images has been enhanced by 400 percent in order to show their properties more clearly.

Table 2 lists some times for combinations of different sorting and rendering techniques. Please note that the qsort based sorting algorithm will slow down significantly for large cluster sizes, as it is $O(n \log n)$ compared to $O(n)$ for the bucketsort. The two bucketsort variants use different bucket sizes, trading speed for quality. The al-

	x	y	z	w
posoffset	-	-	-	-coord quant. offset
basepos	rel. base coords			coords scale
scale	point sprite scale	-	alpha scale	size scale
posin	rel. point coords			point size

```
void main (in      float4 posin   : POSITION,
           in      float4 colin   : COLOR0,
           out     float4 posout  : POSITION,
           out     float4 colout  : COLOR0,
           out     float4 sizeout : PSIZE,
           uniform float  atten,
           uniform float4 posoffset,
           uniform float4 basepos,
           uniform float4 scale)
{
  uniform float4x4 model = glstate.matrix.modelview[0];
  uniform float4x4 proj  = glstate.matrix.projection;
  float4  vec, homeye, eye;
  float   tmp;

  /* relative coords → absolute coords */
  vec.xyz   = (posin.xyz + posoffset.www)
              * basepos.www + basepos.xyz;
  vec.w     = 1.0;
  /* modelview transformation + projection */
  homeye    = mul (model, vec);
  posout    = mul (proj, homeye);
  eye       = homeye / homeye.w;
  /* effective point size calculation */
  tmp       = posin.w * basepos.w * scale.w
              * rsqrt (atten * dot (eye.xyz, eye.xyz));
  /* clamping minimum point size to 1 */
  sizeout.x = scale.x * max (tmp, 1.0);
  /* alpha calculation and attenuation for point sizes < 1 */
  tmp       = min (tmp, 1.0);
  colout    = colin;
  tmp       = colin.w * scale.z * tmp * tmp;
  /* clamping minimum alpha value to keep extremely small points visible */
  colout.w  = max (tmp, 4.0/256);
}
```

Figure 15: The vertex program written in Cg.

gorithm using larger buckets has almost the same visual appearance as the qsort algorithm, but exhibits some flickering during rotation on critical data sets containing large and almost overlapping points.

The cluster selection scheme has about the same performance impact on the rendering system as the flexible rendering backend (less than 2 percent each), which allows us to switch the rendering technique on the fly. Please note that for large viewports like 1000^2 the effect of adaptive hierarchy traversal is not noticeable for low maximum traversal depths, as all clusters are traversed due to their large projected size.

Things change when we reduce the viewport size. Table 3 lists rendering times and the number of rendered points in the levels 1 to 6 for this scheme with no maximum traversal depth. We get early view frustum culling at almost no cost for the adaptive rendering algorithm, as this can be incorporated in the point projection size calculation process. However, all tables show rendering times and point numbers for viewing the full data set.

Figure 16 shows a close up region, rendered intentionally with a very high projection size error of 14 in order to reveal the differences. The next two images show the same region rendered without adaption with approximately the same number of points and all points, respectively. Note that the projected screen size is only an approximation for the maximum screen space error, as the centroid size that used for evaluating the screen space error is not directly coupled with the maximum distribution width of the children which influences the error as well.

For more images and some realtime animations please take a look at our web site:
`http://www.vis.uni-stuttgart.de/pointclouds/`

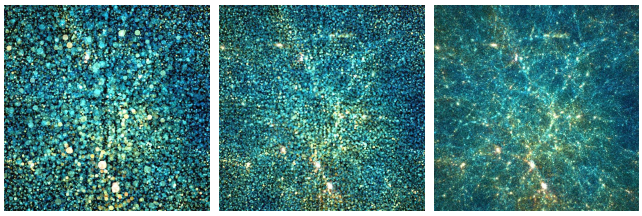


Figure 16: A closeup of the virgo data set, rendered at the really coarse level 3 (123K pts = 0.74%, left), adaptively with approximately the same number of points and high potential projection error (130K pts, middle), and with all points (16.8M pts, right), respectively.

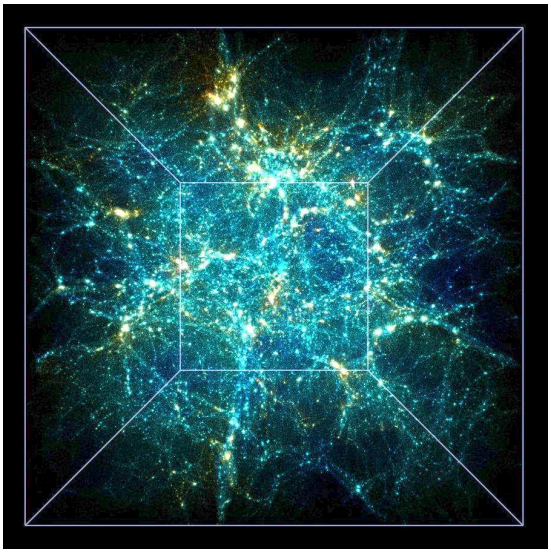


Figure 17: A total view of one of the close up simulations with 16.8 million particles as well.

10 Conclusion and Future Work

In this paper we presented a technique for accelerating the visualization of scattered point data compared to rendering flat point arrays. We employed principal component analysis for creating a hierarchy of point clusters, stored with quantized relative coordinates in a data structure that separates cluster from point data. With this data representation visualization quality can be traded for speed with an adaptive rendering algorithm. The rendering process itself was accelerated using vertex programs on modern PC graphics hardware. Finally, we are now able to visualize data sets with tens of millions of points interactively on standard workstations.

One of the most promising — but also most challenging — extensions to the algorithm is the handling of time dependent data. The rendering process can be left almost unchanged, but it is a topic of ongoing research how the clustering step can be improved. It will have to handle cluster transitions of single particles in a smooth way, such that popping artifacts will not occur.

The current implementation of the clustering algorithms requires all points as floating point data in memory, but they can be performed on some other, remote supercomputer. Out-of-core clustering algorithms exist, and we want to investigate how they can be applied to our case. Additionally, there are some issues with the overestimation of the radiant flux during rendering with cumulative blending in areas of high saturation. The system should detect these areas and reduce the brightness of the generated clusters accordingly. Alternatively, high dynamic range rendering to floating point frame buffers could be used.

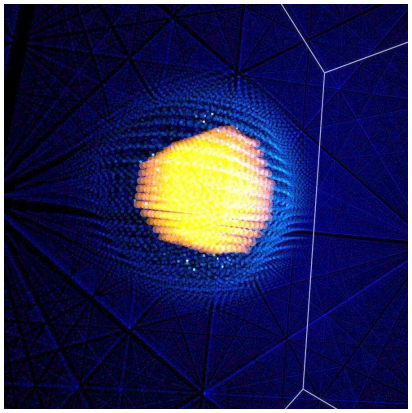


Figure 18: Visualization of a shock front, simulated with SPH (1 million points).

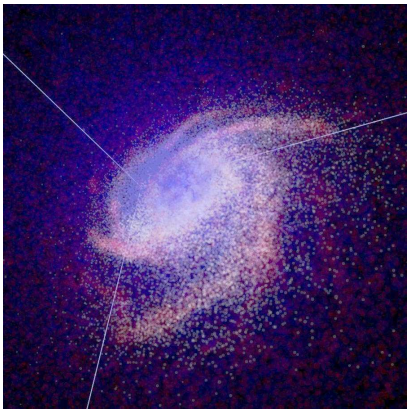


Figure 19: SPH and dark matter galaxy formation simulation rendered with sorted anti-aliased points (540,000 points).

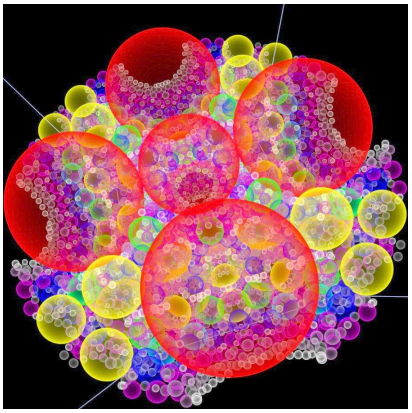


Figure 20: Reversible Apollonian packing rendered with sorted point sprites.

An open question is how to handle multivariate data and how to change the visualized data during runtime. Storing several color values per point structure is one possibility, but this increases memory usage again. Color quantization and table lookup in the rendering step could help with regard to this aspect.

There are still some issues with the rendering of sorted points with respect to overlapping points in adjacent clusters. Additionally, there might be a chance to implement bucket sorting by rendering to off-screen textures with the next generation graphics hardware, and radix sorting could be an interesting alternative as well.

11 Acknowledgments

This work has been financed by the project SFB 382 of the German Research Foundation (DFG).

Some of the data sets shown in this paper are based on simulations carried out by the Virgo Supercomputing Consortium using computers based at the Computing Center of the Max-Planck Society in Garching and at the Edinburgh Parallel Computing Center. The data sets are publicly available at `www.mpa-garching.mpg.de/NumCos/`.

We would like to thank Sebastian Niedworok (TAT) from the University of Tübingen, Volker Springel and Martin Jubelgas from the Max-Planck Society in Garching, as well as Reza Mahmoodi Baram (ICA) and Johannes Roth (ITAP) from the University of Stuttgart for their support with data sets and insight into their visualization needs. Additionally, we would like to thank our colleagues Marcelo Magallón, Guido Reina, and Daniel Weiskopf for helpful discussion.

References

ALEXA, M., BEHR, J., COHEN-OR, D., FLEISHMAN, S., LEVIN, D., AND SILVA, C. T. 2001. Point Set Surfaces. In *Proc. Visualization '01*, IEEE, 21–28.

BOTSCH, M., WIRATANAYA, A., AND KOBBELT, L. 2002. Efficient High Quality Rendering of Point Sampled Geometry. In *Proc. Eurographics Workshop on Rendering '02*, EG.

COX, D. J. 1996. Cosmic Voyage: Scientific Visualization for IMAX film. In *SIGGRAPH 96 Visual Proceedings*, ACM, 129,147.

GAIA. 2003. *ESA's space astrometry mission*. http://astro.estec.esa.nl/GAIA/gaia.html.

JANG, J., RIBARSKY, W., SHAW, C. D., AND FAUST, N. 2002. View-Dependent Multiresolution Splatting of Non-Uniform Data. In *Proc. Eurographics VisSym '02*, IEEE.

JENKINS, A., FRENK, C. S., PEARCE, F. R., THOMAS, P. A., COLBERG, J. M., WHITE, S. D. M., COUCHMAN, H. M. P., PEACOCK, J. A., EFSTATHIOU, G. P., AND NELSON, A. H. 1998. Evolution of Structure in Cold Dark Matter Universes. *ApJ 499*, 1, 20–40.

JOLLIFFE, I. T. 1986. *Principal Component Analysis*. Springer, New York.

KÄHLER, R., COX, D., PATTERSON, R., LEVY, S., HEGE, H.-C., AND ABEL, T. 2002. Rendering The First Star In The Universe - A Case Study. In *Proc. Visualization 02*, IEEE, 537–540.

LAUR, D., AND HANRAHAN, P. 1991. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. In *Proc. SIGGRAPH '91*, ACM, 285–288.

MARK, W. R., GLANVILLE, S., AND AKELEY, K. 2003. Cg: A System for Programming Graphics Hardware in a C-like Language. In *Proc. SIGGRAPH '03*, ACM.

MAX, N., HANRAHAN, P., AND CRAWFIS, R. 1990. Area And Volume Coherence For Efficient Visualization Of 3D Scalar Functions. *ACM Computer Graphics (San Diego Workshop on Volume Visualization) 24*, 5, 27–33.

MONAGHAN, J. J. 1992. Smoothed Particle Hydrodynamics. *Ann. Rev. Astron. Astrophys. 30*, 543–574.

MUELLER, K., MÖLLER, T., AND CRAWFIS, R. 1999. Splatting without the Blur. In *Proc. Visualization '99*, IEEE, 363–370.

NADEAU, D. R., GENETTI, J. D., NAPEAR, S., PAILTHORPE, B., EMMART, C., WESSELAK, E., AND DAVIDSON, D. 2000. Visualizing Stars and Emission Nebulas. In *Proc. EUROGRAPHICS '00*, Eurographics Association.

PARK, S., BAJAJ, C., AND SIDDAVANAHALLI, V. 2002. Case Study: Interactive Rendering of Adaptive Mesh Refinement Data. In *Proc. Visualization 02*, IEEE Computer Society Press, IEEE Computer Society, 521–524.

PAULY, M., GROSS, M., AND KOBBELT, L. 2002. Efficient Simplification of Point-Sampled Surfaces. In *Proc. Visualization '02*, IEEE, 163–170.

PFISTER, H., ZWICKER, M., VAN BAAR, J., AND GROSS, M. 2000. Surfels: Surface Elements as Rendering Primitives. In *Proc. SIGGRAPH '00*, ACM, 335–342.

PUGH, W. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM* (June).

RAU, R., AND STRASSER, W. 1995. Direct Volume Rendering of Irregular Samples. In *Visualization in Scientific Computing '95*, R. Scateni, Ed., 72–80.

REZK-SALAMA, C., ENGEL, K., BAUER, M., GREINER, G., AND ERTL, T. 2000. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. In *EG/SIGGRAPH Workshop on Graphics Hardware '00*, ACM, 109–118,147.

RUSINKIEWICZ, S., AND LEVOY, M. 2000. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proc. SIGGRAPH '00*, ACM, 343–352.

SWAN, J. E., MUELLER, K., MÖLLER, T., SHAREEF, N., CRAWFIS, R., AND YAGEL, R. 1997. An Anti-Aliasing Technique for Splatting. In *Proc. Visualization '97*, IEEE, 197–204.

WAND, M., FISCHER, M., PETER, I., MEYER AUF DER HEIDE, F., AND STRASSER, W. 2001. The Randomized z-Buffer Algorithm. In *Proc. SIGGRAPH '01*, ACM, 361–370.

WILLIAMS, P. L. 1992. Visibility Ordering Meshed Polyhedra. *ACM Transactions on Graphics 11*, 2, 103–126.

ZWICKER, M., PFISTER, H., VAN BAAR, J., AND GROSS, M. 2001. EWA Volume Splatting. In *Proc. Visualization '01*, IEEE, 29–36.

ZWICKER, M., PFISTER, H., VAN BAAR, J., AND GROSS, M. 2001. Surface Splatting. In *Proc. SIGGRAPH '01*, ACM, 371–378.