In *Proceedings IASED VIIP Conference*, pages 628–633, 2005.

XSplat: External Memory Multiresolution Point Visualization

Renato Pajarola[*]
[*]Visualization and MultiMedia Lab
Department of Informatics
University of Zürich

Miguel Sainz[†]
[†]NVIDIA Corporation

Roberto Lario[‡]
[‡]Dpto. Arquitectura de
Computadores y Automática
Universidad Complutense Madrid

Abstract

With the popularity of points as graphics primitives, it is important to handle large-scale point sets that exceed available in-core (main) memory. In particular, high-performance level-of-details (LODs) visualization from out-of-core is a challenging problem. In this context we present a novel point-splatting approach, short XSplat, that breaks the main memory barrier. It is based on a paginated multiresolution point hierarchy and virtual memory mapping. The main contributions are a novel block-based sequential multiresolution point hierarchy, an efficient LOD-block paging mechanism and dynamic mapping into video-cache. XSplat is scalable by using sequentialized data structures, and it seamlessly bridges the disk-, main- and video-memory sub-systems. Experiments demonstrate the quality and efficiency that is achieved by XSplat.

Key Words: out-of-core visualization, point-based rendering, multiresolution, levels-of-detail, geometry caching

1. Introduction

Points as display primitives have become a powerful alternative to traditional 3D object representations. In fact, points or 3D coordinates are the most simple and fundamental geometry-defining entities. Discrete point samples have long been established in the field of volume rendering as splatting primitives. Points have only recently received increased attention (i.e. after [17]) in the context of surface representations. Furthermore, points have proved very useful in different aspects of visualization such as multiresolution modeling [2, 6, 9, 12], simplification [26], and for example rendering of surface uncertainty [16] or scattered data [18].

With the dramatically increasing data sizes, it has become increasingly difficult to visualize the generated large point sets in an efficient way. Level-of-detail (LOD) techniques [23] trade-off object complexity and accuracy for rendering performance. However, standard multiresolution techniques fail with models exceeding the physical main memory capacity due to uncontrolled memory-trashing from random virtual memory accesses, possibly causing the system to almost come to a halt. To cope with this situation, efficient out-of-core based multiresolution techniques are necessary.

Contributions: In this paper we introduce *XSplat* that fills the gap of out-of-core multiresolution point-representation and interactive-visualization. The main contributions are an out-of-core multiresolution data structure and a LOD-based visualization algorithm based on the following novel concepts:

- two-way interleaved sequential ordering of a space-subdividing multiresolution point-hierarchy,
- paginated organization of point data to reduce overhead for micro-management of huge data sets,
- paging from out-of-core to in-core memory as well as dynamic mapping into video-memory.

Figure 1: Example visualizations of large point data from out-of-core using XSplat. Lucy displayed using about 5M and David 3.4M points. From out-of-core, Lucy renders at about 1.2 fps and David at about 3 fps.

2. Related Work

Point-based splatting has widely been used in volume visualization. However, points scattered on surfaces pose an entirely different problem setting with their irregular sampling and distribution in 3D space. In particular we target multiresolution modeling and interactive visualization of large point-sampled surface data sets from out-of-core, external memory.

Closely related works are point-based techniques for efficient multiresolution modeling and view-dependent LOD-based rendering. Very efficient point-based hierarchical multiresolution representations have been proposed with a focus on efficient representation [29, 5], hardware accelerated rendering [29, 28, 3, 25], or integration with polygons [6, 9, 12, 8]. Other point-based approaches have concentrated more on high-quality rendering [27, 34, 19, 4]. See also the overviews [31, 30] for more on PBR. Our work differs significantly in that we introduce a novel paginated and sequential out-of-core multiresolution data structure for view-dependent LOD visualization.

An efficient method to render points in a sequential way has been proposed in [10]. Note that in [10] the entire LOD point-hierarchy generally has to fit in video memory and is thus not directly applicable to data exceeding physical main memory or even graphics card video memory. Furthermore, this approach does not allow for any visibility culling before caching and GPU processing. Our approach is similar in that we use a sequential data arrangement, however, we take it to the next level. We employ two different interleaved sequential orderings: one in space for individual points and one in the LOD-metric for blocks. Moreover, XSplat takes all memory levels of the system into account for caching data and allows for visibility culling.

[*]pajarola@acm.org, [†]msainz@ics.uci.edu, [‡]rlario@dacya.ucm.es

In [15] a hierarchy of point-cloud blocks is proposed for efficient rendering of very large point data sets which offers simple progressive block refinement and culling, exploits caching in graphics memory, and hides disk latency effectively. We discuss the differences between this approach and ours in Section 6.

A number of out-of-core techniques have appeared for simplification and multiresolution rendering of massive polygonal meshes. In particular, triangle meshes have been addressed in [13, 21, 11, 7, 22] and general polygonal models in [32]. These techniques are fundamentally different in that face-connectivity is an integral part of the mesh representation which must be maintained.

3. Out-of-Core Multiresolution Modeling

3.1 Level-of-Detail Hierarchy

The data is assumed to be a set of 3D points $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n \in \mathbf{R}^3$ that satisfy necessary surface sampling criteria such as the Nyquist sampling condition, and fully define the geometry as well as the topology of a surface. Furthermore, it is assumed that the points are initially organized in a LOD-hierarchy as outlined below.

In a preprocess, XSplat converts a conventional point-based LOD-hierarchy to a sequential out-of-core multiresolution data structure. This input can be any nested point-hierarchy such as the widely used point-octree variants (i.e. [29, 5, 3, 24, 10, 25]). The nesting properties include bounding sphere and normal-cone attributes in each node, which confine all elements and normals in the corresponding subtree. Figure 2 illustrates the LOD-node format of such a multiresolution point-hierarchy H which is embedded as an array of node elements. Each node H_i represents a disk with radius r_i and color $\boldsymbol{c}_i$ centered at position $\boldsymbol{p}_i$ and oriented with respect to normal $\boldsymbol{n}_i$.

LOD-node i
average position $\boldsymbol{p}_i$
average normal $\boldsymbol{n}_i$
average color $\boldsymbol{c}_i$
bounding sphere radius r_i
bounding normal cone semi-angle θ_i
index to first child node $first_i$
number of child nodes n_i

Figure 2: Node attributes of a nested LOD-hierarchy H.

The LOD-metric implemented in XSplat is a commonly used screen-space area error. Given the viewpoint $\boldsymbol{v}$ and a user specified screen-space tolerance ε, in a top-down traversal of H a LOD-node i is rendered if $\pi \cdot r_i^2 \cdot |\boldsymbol{p}_i - \boldsymbol{v}|^{-2} \le \varepsilon$ and refined otherwise if not a leaf node. The bounding sphere and normal-cone attributes allow for effective visibility culling in a LOD-selection algorithm. Given the normals $N_{1..4}$ of a four-sided view-frustum pyramid and the viewpoint $\boldsymbol{v}$, a node i is outside the view-frustum if $(\boldsymbol{p}_i - \boldsymbol{v}) \bullet N_j > r_i$ for any of the normals $N_{1..4}$. A node i is back-face culled if the angle between $(\boldsymbol{v} - \boldsymbol{p}_i)$ and normal $\boldsymbol{n}_i$, minus θ_i, is larger than 90°.

3.2 Sequential Layout

For efficient out-of-core management, XSplat converts the LOD-hierarchy H into a sequential list of points S. For this we must resolve the LOD-refinement dependencies in the hierarchy H as described in [10]. We can define the minimal distance $rmin_i$ at which a node i will be split for a given error tolerance ε by $\varepsilon \cdot rmin_i^2 = \pi \cdot r_i^2$. Consequently we can define a maximal distance $rmax_i$ at which the node i will be merged based on the $rmin_j$ of its parent node j. This, however, must be compensated for the distance between node i and its parent j to arrive at a conservative measure for all possible viewpoints. Hence we get a merge distance of $rmax_i = rmin_j + |\boldsymbol{p}_i - \boldsymbol{p}_j|$. The point attributes of elements in this sequential LOD point-sequence S are given in Figure 3.

Render-point i
average position $\boldsymbol{p}_i$
average normal $\boldsymbol{n}_i$
average color $\boldsymbol{c}_i$
bounding sphere radius r_i
split distance $rmin_i$
merge distance $rmax_i$

Figure 3: Point attributes of a sequentialized hierarchy S.

Given a viewpoint $\boldsymbol{v}$ and a user specified screen-space tolerance ε, all points i of S with $rmin_i \le \sqrt{\varepsilon} \cdot |\boldsymbol{p}_i - \boldsymbol{v}| \le rmax_i$ are selected for rendering. Note that as discussed in [10] this LOD selection is conservative as for some nodes j also some descendant nodes i — with respect to the initial hierarchy H — may be rendered as well. However, this does not affect the LOD selection and visualization efficiency noticeably in practice.

The major feature of this point-sequence S is that the LOD selection has been de-coupled from the hierarchical data structure that holds the points and guides the tree traversal. This fact is exploited in [10] to sort all points in S with respect to their *rmax* values which allows a fast, coarse-grain and conservative LOD selection of a range of sequential points. All points within this range are then submitted to the graphics hardware which does the fine-grain LOD selection as outlined above. Note that no visibility culling can so be performed before the entire conservative range of points is processed by the graphics engine. Here is where XSplat differs significantly to accommodate for effective out-of-core multiresolution modeling and visualization.

We first observe that a recursive spatial subdivision hierarchy implicitly defines a space-filling curve index on the nodes. In fact, a proper traversal order defines a hierarchical z-order (see also [14]) which preserves spatial locality and thus improves coherence in memory access. This index is illustrated in Figure 4. Note that we consider all n leaf nodes to be consecutively numbered from z_j to z_{j+n-1}. In a multiresolution context this linear index must be combined with the LOD-metric, e.g. by a top-down *level*-wise grouping (classification) of the nodes. This can lead to grouping nodes of significantly different LOD into the same level as shown in Figure 4, where level 2 contains LOD-nodes that contain from 1 up to 6 original input points (leaf nodes). Hence subtrees rooted in the same level can differ dramatically in point coverage and thus also in the LOD-metric.

Considering the bounding sphere size which is used in the LOD-metric, a bottom-up LOD classification is much more likely to put nodes of similar LOD-importance into the same class. Therefore, we classify the nodes of the input hierarchy H based on a *layer*-index as shown in Figure 4. This layer number l_i is basically the length of the longest path from a node i to any leaf in its subtree. Second, we set z_i of each (inner) node i to the smallest z_j of any of its descendants (leafs) to preserve the spatial ordering within all layers and levels.

Each point/node from H thus is associated with an index pair (l_i, z_i) consisting of layer and spatial ordering information. To arrive at the final ordering of the sequence of render-points S_i (see Figure 3) corresponding to nodes in H_i, we order them lexicographically in (l_i, z_i), decreasing in l_i and increasing in z_i, as illustrated in Figure 5. All render-points of layer l will so be stored consecutively in the sequence S.

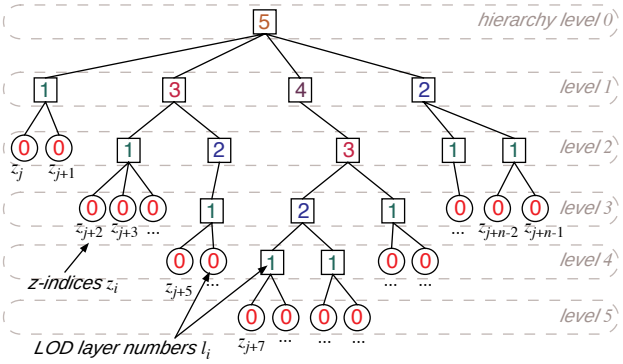

Figure 4: Hierarchical *level*- and *layer*-based LOD classification, and linear ordering of leaf nodes in *z*-index.

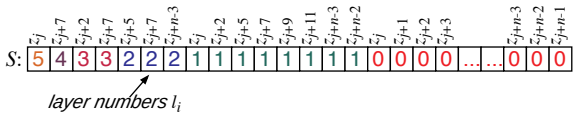

Figure 5: Layer-wise ordering of LOD render-points in sequential LOD representation *S*.

3.3 Pagination

Since a fine-grain visibility culling, LOD selection, paging and caching on the point sequence *S* is not feasible for large data, in particular when considering accessing data from out-of-core, XSplat paginates the point sequence *S* into a list of blocks *B* as depicted in Figure 6. The pagination starts at the first layer *l* with more elements than block capacity. Also, the layers are padded by NULL-points to an integral multiple of the the block capacity. Each block *i* references its points by an index $first_i$ to the first element in *S* and the number n_i of elements in block B_i.

As the number *m* of blocks in the block list *B* is an order of magnitude smaller than the number *n* of render-points, an efficient coarse culling, LOD selection, paging and caching can be performed on the set *B* as described in more detail in Section 4.

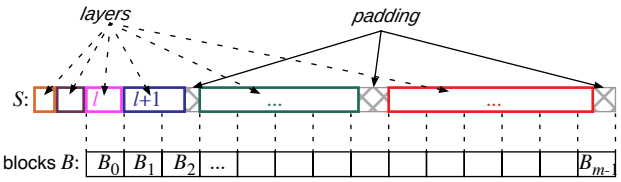

Figure 6: Pagination and padding into blocks of point-sequence *S*.

For effective visibility culling, each block B_i stores bounding sphere radius r_i and normal-cone semi-angle θ_i attributes over all points S_i in B_i, see also Figure 7 for the render-block node format. Therefore, as for individual points, it holds for a block B_i, with center $\boldsymbol{p}_i$, normal $\boldsymbol{n}_i$, bounding sphere radius r_i and normal-cone semi-angle θ_i, that: given a viewpoint $\boldsymbol{v}$ and view-frustum normals $N_{1..4}$, B_i is not visible and hence culled if $(\boldsymbol{p}_i - \boldsymbol{v}) \bullet N_j > r_i$ or $\omega - \theta_i > 90°$, for the angle ω between $(\boldsymbol{v} - \boldsymbol{p}_i)$ and normal $\boldsymbol{n}_i$.

Furthermore, as for the point sequence *S*, the block list *B* has no more a hierarchical organization. However, the blocks *B* can exploit the same concept of split and merge distances *rmin* and *rmax* of points in *S* on a per-block basis in *B*. For a block B_i and all of its points $j; S_j \in B_i$ we set $rmin_i = \mathrm{MIN}_j(rmin_j)$ and $rmax_i = \mathrm{MAX}_j(rmax_j)$. Hence given the viewpoint $\boldsymbol{v}$ and screen-space tolerance ε, a block B_i contains rendered points only if it holds that $rmin_i \le \sqrt{\varepsilon} \cdot (|\boldsymbol{p}_i - \boldsymbol{v}| + r_i)$ and $rmax_i \ge \sqrt{\varepsilon} \cdot (|\boldsymbol{p}_i - \boldsymbol{v}| - r_i)$. The addition and subtraction of the block's bounding sphere radius r_i accounts for the possible spatial deviation of points within a block, as indicated in Figure 8.

Render-block *i*

- average position $\boldsymbol{p}_i$
- average normal $\boldsymbol{n}_i$
- bounding sphere radius r_i
- bounding normal cone semi-angle θ_i
- split distance $rmin_i$
- merge distance $rmax_i$
- index to first point $first_i$
- number of points n_i

Figure 7: Block attributes of a paginated sequential point-list *S*.

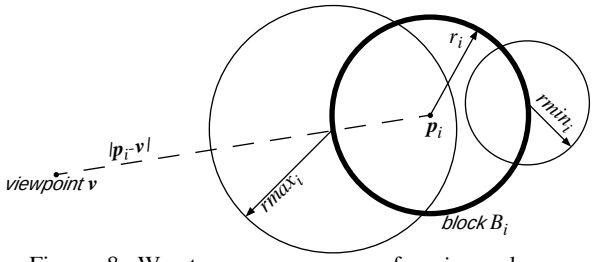

Figure 8: Worst-case occurrence of $rmin_i$ and $rmax_i$ within a block B_i.

To aid fast estimation of blocks to render for a given viewpoint $\boldsymbol{v}$, we apply the sorting proposed in [10] to blocks. Hence the blocks B_i are ordered according to their $rmax_i$ value as depicted in Figure 9. Based on the largest bounding sphere of block 0 (radius r_0, center $\boldsymbol{p}_0$) the merge and split distances are $dmin = \sqrt{\varepsilon} \cdot (|\boldsymbol{p}_0 - \boldsymbol{v}| + r_0)$ and $dmax = \sqrt{\varepsilon} \cdot (|\boldsymbol{p}_0 - \boldsymbol{v}| - r_0)$. Within the *rmax*-ordered block list *B*, only the conservative range [*lo*, *hi*] must be considered for rendering as shown in Figure 9; with *lo* and *hi* being the smallest and largest index for which $rmin_{lo} < dmin$ and $rmax_{hi} > dmax$. Further details on fast LOD-block selection for rendering is given in Section 4.

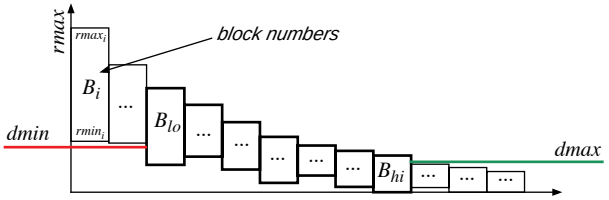

Figure 9: Ordering of blocks in *B* with respect to the block's *rmax* values and selection of range.

3.4 File Format

As described in the preceding sections, the XSplat data structures consist of an array *S* of render-points (Figure 3) and an array of blocks *B* (Figure 7). Both of these arrays are computed in a pre-process. (See also Section 3.1) Most preprocessing can be done also using memory mapped files.

The input hierarchy *H* must be traversed only once to generate the layer- and space-index pairs (l_i, z_i) for each render-point S_i as well as the split/merge distances $rmin_i$/$rmax_i$ described in Section 3.2. This resulting array *S*, and its temporary indices (l_i, z_i), can be maintained in a memory mapped (read-write) file for out-of-core processing. The array *S* is then sorted by the imposed index order (l_i, z_i). This sorting can be achieved by a quicksort algorithm on *S*, which also performs well since quicksort exhibits strong memory access coherence.

The (l_i, z_i)-ordered and memory mapped array S is the XSplat rendering data format.

The block array B is initialized by one linear pass over the render-points S generating the block attributes (Section 3.3). This is followed by a sorting of B in *rmax*. As the size of B is an order of magnitude smaller than S it can generally be managed in main memory, but as well using a memory mapped file out-of-core.

In the current implementation, the arrays B and S are in fact consecutively stored as two segments in one and the same *single binary file* with some additional header information such as number of elements. The XSplat rendering tool reads and memory maps the file accordingly to access blocks and render-points.

4. Rendering

4.1 Overview

For efficient LOD-visualization it is critical to quickly arrive at a reduced object complexity for the given view. XSplat performs this LOD reduction based on block information (Section 3.3). The coarse-grain block-based LOD selection and visibility culling uses minimal CPU time to reduce the complexity as much as possible. At the expense of little CPU time block-based LOD and culling can improve over a pure sequential point-range selection [10] by reducing the data submitted to the GPU. In particular, it allows for visibility culling on a block-level not possible otherwise. Paginated processing also reduces CPU cost compared to a top-down traversal of the multiresolution hierarchy with per-vertex LOD-selection and visibility-culling. Block-based processing can strike a better CPU/GPU load-balance where coarse filtering is performed on the CPU and fine-grain evaluation on the GPU.

Block-based culling and LOD-evaluation becomes even more important for large point sets that exceed available video and main memory size. Per-point evaluation becomes infeasible as the large number of individually processed points can clog the CPU and may cause excessive paging of virtual memory. Conservative selection of point-ranges curbs caching in video memory, as a large fraction of points is wasted and floods the bus and GPU with elements which will be rejected and not rendered.

The three main stages are described in the following sections:

1. In the first stage, a coarse LOD-selection and visibility culling on the blocks B is performed on the CPU to quickly cut down the data size for rendering the object with respect to the current viewing parameters.
2. If not already in the geometry cache, the blocks selected for rendering are copied to the video memory.
3. The visible blocks are streamed to the GPU which performs the per-point culling and LOD-selection, and image synthesis.

The file-segments holding the render-points S and render-blocks B are memory-mapped to provide seamless out-of-core access via indexing and virtual memory addressing.

4.2 LOD Selection and Culling

When graphics card video memory is used to cache geometric data, the number of rendered points per frame is limited to a constant of C (blocks). Section 4.3 explains the dynamic updating of this geometry cache. The basic block selection is performed as in Section 3.3 and illustrated in Figure 9, however, must be adjusted to select at most C blocks. Blocks and their render-points can also directly be rendered from the memory mapped array S without caching if desired.

The conservative $dmin = \sqrt{\varepsilon} \cdot (|\boldsymbol{p}_0 - \boldsymbol{v}| + r_0)$ and $dmax = \sqrt{\varepsilon} \cdot (|\boldsymbol{p}_0 - \boldsymbol{v}| - r_0)$ give an initial range $[lo, hi]$ on the array B, a starting point to select at most C blocks. The hi range is easily found in the ordered array B by binary search for $rmax_{hi}$ just above $dmax$. The lower bound lo is found by a linear search from the start of B.

The conservative range $[lo, hi]$ is then scanned as shown in Figure 10 to select up to C blocks. A per-block B_i error tolerance $\varepsilon_i = (rmax_i / (|\boldsymbol{p}_i - \boldsymbol{v}| - r_i))^2$ is maintained for block center $\boldsymbol{p}_i$ and radius r_i to define the achieved error threshold when rendering is limited to C blocks. The cache size C is targeted by starting with $\varepsilon_{cur} = \varepsilon_{lo}$ and continually adding the next block B_i with the next smaller $\varepsilon_i \le \varepsilon_{cur}$. Thus decreasing the tolerance ε_{cur} until the cache size C is met. Outside-view and back-face visibility culling of blocks can be accounted for while scanning the blocks due to fixed viewing parameters for a given frame. Culled blocks do not waste any slots in the geometry cache. If no cache is used, the visible blocks within $[lo, hi]$ that pass the LOD-evaluation are rendered.

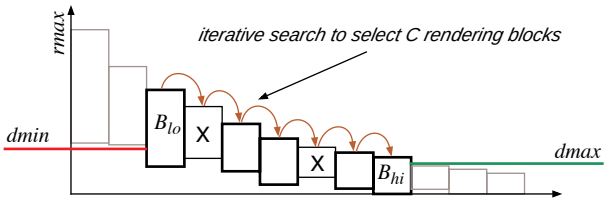

Figure 10: Scanning for C number of blocks in B, that fit into geometry cache if desired, within the conservative range $[lo, hi]$.

4.3 Geometry Caching

XSplat can use video memory to cache the geometry of render-points, but it can also visualize large out-of-core data without caching as demonstrated in the experiments of Section 5. The format to access video memory is via vertex array ranges. In fact, the binary layout of the render-point format given in Figure 3 is such that it can directly be used by the OpenGL graphics API as interleaved vertex arrays.

The geometry cache manager partitions the available video memory in C slots to hold compact vertex array ranges corresponding to blocks. If cached, the entire vertex array S_{first_i} to $S_{first_i + n_i - 1}$ of a block B_i is copied as a whole from the memory-mapped array S to a video memory slot, see also Figure 11. The cache manager keeps track of re-usable slots by a least-recently-used strategy. Each of the C slots features a timestamp when its content was last rendered on screen. The oldest slot is overwritten if new data needs to be fitted in the cache.

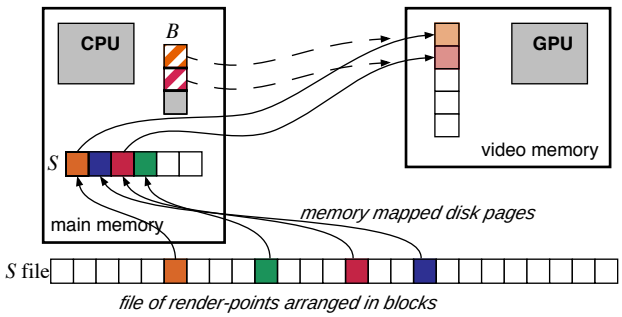

Figure 11: Organization of render-point file S as memory-mapped array, and block-array B in conjunction with optional geometry caching in video memory.

4.4 Point Splatting

The back-end point-splatting pipeline is similar to other recent approaches. In particular, XSplat offers the choice between GLPoints, opaque sprites, and α-textured sprites. The sprites represent the surface by oriented circular disks while the GLPoints only draws screen aligned splats. The α-textured sprites provide a smooth and continuous poing blending. See [30] and [31] for more details on these rendering primitives.

As in [10] a fine-grain visibility culling and LOD-evaluation is done on the GPU over the individual points *Sj* in the list of blocks B_i selected by the main CPU culling and LOD-selection algorithm outlined above. The visibility culling and LOD-simplification as outlined in Figure 3 is performed by a vertex-program.

5. Experiments

The experiments reported in this section were performed on a Dell Pentium4 PC with 2.4GHz CPU, 512MB main memory and NVIDIA GeForce 5900 GPU.

In Table 1 we summarize the test models. The columns .spt and .blk denote the sequential point tree format [10] (.spt) and XSplat's block-based out-of-core format of Section 3 (.blk). The .blk format contains both, the ordered array of blocks *B* and the two-way sorted sequential points *S*. The block capacity has been set to 64 render-points per block.

Model	#points	.spt	.blk
Lucy	14,022,961	760MB	770MB
David 2mm	4,129,534	224MB	230MB
David head	2,000,646	109MB	112MB
Female	302,948	16MB	16.5MB

TABLE 1. Test models.

In Table 2 we report timings of XSplat. It reports ammortized values of rendering 1000 frames at 512x512 image resolution flying around the object with just about the full object in view and rendered as GLPoints (e.g. see Figure 1 on page 1). The second column denotes the screen projection error tolerance in percentage of the viewport size. The third column shows the average number of visible points. This represents the CPU block-processed coarse approximation. The fourth column shows the number of rendered points per second (PPS) with respect to the overall cost (including visibility culling, LOD selection and rendering) and just rendering. Note that David and David head at 0.0001 used geometry caching on the video memory, hence the high FPS achieved.

Model	Tol. ε	#pts/frame	#pts/sec	FPS
Lucy	0.0%	5.1M	5.9M / 9.3M	1.2
David 2mm	0.0%	3.4M	9.6M / 12.3M	2.9
	0.0001%	663K	11.1M / 1.2G	17
David head	0.0%	2.3M	10M / 12.4M	4.5
	0.0001%	467K	12.3M / 1.2G	26
Female	0.0%	839K	12M / 0.4G	36

TABLE 2. Visualization experiments averaged over 1000 frames. Performance in points per second given for overall timing and for rendering only.

Performance of the out-of-core memory mapped file is presented in Table 3. The peak-values are reported by Windows XP Task Manager during the execution. The second column presents the *Working Set* of the application, process memory and shared memory with other processes. The third column reports the exclusive memory allocated by the process. We can observe the effectiveness based on the David 2mm and Lucy models which only partially map the out-of-core data into main memory: The performance, despite the out-of-core mapping and memory management, scales well compared to the smaller David head model which fully resides in-core.

Model	Mem Usage(KB)	Virtual(KB)	File Size (KB)
Lucy	490,028	28,408	776,379
David 2mm	136,992	14,184	229,648
David head	118,592	11,624	111,787
Female	22,420	7,524	16,930

TABLE 3. Peak memory allocation for the experiments.

6. Discussion

As shown in Section 5, large models accessed from out-of-core at run-time (see Table 3) can be rendered by XSplat at several million PPS (see Table 2). The achieved FPS are generally lower than in-core rendering as reported in [30]. Besides the approach in [15], no previous point-rendering systems is targeted at interactive rendering from out-of-core.

In [15], *layered point clouds* (LPC) for rendering huge point sets have been proposed. Indeed LPC is very efficient in that its block-based hierarchy is simple to traverse for LOD-selection or culling and caching in video-memory is supported. An important difference is that LPC heavily exploits quantization of geometric attributes. This results in very large models occupying not more than a few 100MB in memory, and thus fitting into main memory. The rendering experiments in [15] were performed with 2GB main memory which can keep ***all*** of their test models in-core. In contrast, our tests were performed with limited memory to explore real out-of-core access (i.e. for the Lucy model). LPC [15] reports average rendering rates of 30M to 40M splats/sec. However, it is not declared if these numbers are derived from the full model size multiplied by the frame rate, or if only the final selected points are considered. XSplat achieves 6M to 12M visible and displayed splats/sec including culling, LOD-selection and rendering; or 9M to 1,200M points/sec considering the rendering part only.

Due to their differences, we believe a quantitative comparison of XSplat to LPC is out-of-scope in this place. However, we acknowledge the benefit of quantization as applied in LPC which allows many more points to be kept in-core per memory unit. This greatly reduces the overall amount of memory to be accessed for rendering, which not only noticeably improves display performance but also significantly decreases out-of-core access time. The arguable performance benefit of LPC over XSplat potentially stems from this memory access difference due to quantization. Hence reducing memory usage based on rigorous quantization is a viable approach in general, and for future extensions of XSplat.

Figure 12: A close-up image of the David head model using the high quality blended point rendering system.

7. Conclusion

We have presented a novel out-of-core point visualization algorithm. The main advantages: simplicity of its sequential data layout, no quantization or compression of attributes, balanced CPU/GPU load and dynamic and seamless paging from out-of-core to main memory, and graphics video memory. XSplat achieves good performance on large data sets, it renders the large models from out-of-core at the best resolution at more than one FPS. XSplat is also reasonably efficient for medium sized model and offers high-speed rendering when the visible data fits into the geometry cache. In that case, XSplat is able to render 12 million PPS from out-of-core for the 2M point David head model. It even achieves rates of 6 to 9 million points per second from out-of-core. Therefore, it offers a transparent and efficient way of rendering any sized point models on any available main memory configuration.

Given a linearized multiresolution point-hierarchy a simple preprocess – adaptable to out-of-core as well, using memory mapping – has been presented. Further preprocessing steps on how to generate a multiresolution hierarchy and LOD attributes initially are beyond the scope of this paper. However, using external memory sorting techniques (see also [1, 33]) and clever use of memory mapped files, this can be addressed as well.

Acknowledgements

We would like to thank the Stanford *3D Scanning Repository* and *Digital Michelangelo* projects as well as *Cyberware* for freely providing geometric models to the research community. This research was partly supported by awards UCI SIIG-2003-2004-19 and New Del Amo UCDM-33657 .

References

[1] J. Abello and J. S. Vitter. *External Memory Algorithms*. American Mathematical Society, Providence, R.I., 1999.

[2] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shackar Fleishman, David Levin, and Claudio T. Silva. Point set surfaces. In *Proceedings IEEE Visualization*, pages 21–28. Computer Society Press, 2001.

[3] Mario Botsch and Leif Kobbelt. High-quality point-based rendering on modern GPUs. In *Proceedings Pacific Graphics*, pages 335–343. IEEE, Computer Society Press, 2003.

[4] Mario Botsch, Michael Spernat, and Leif Kobbelt. Phong splatting. In *Proceedings Symposium on Point-Based Graphics*, pages 25–32. Eurographics, 2004.

[5] Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. Efficient high quality rendering of point sampled geometry. In *Proceedings Eurographics Workshop on Rendering*, pages 53–64, 2002.

[6] Baoquan Chen and Minh Xuan Nguyen. POP: A hybrid point and polygon rendering system for large data. In *Proceedings IEEE Visualization*, pages 45–52, 2001.

[7] Paolo Cignoni, Claudio Montani, C. Rocchini, and Roberto Scopigno. External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):525–537, October 2003.

[8] Liviu Coconu and Hans-Christian Hege. Hardware-oriented point-based rendering of complex scenes. In *Proceedings Eurographics Workshop on Rendering*, pages 43–52, 2002.

[9] Jonathan D. Cohen, Daniel G. Aliaga, and Weiqiang Zhang. Hybrid simplification: Combining multi-resolution polygon and point rendering. In *Proceedings IEEE Visualization*, pages 37–44, 2001.

[10] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. In *Proceedings ACM SIGGRAPH*, pages 657–662. ACM Press, 2003.

[11] Christopher DeCoro and Renato Pajarola. XFastMesh: Fast view-dependent meshing from external memory. In *Proceedings IEEE Visualization*, pages 363–370. Computer Society Press, 2002.

[12] Tamal K. Dey and James Hudson. PMR: Point to mesh rendering, a feature-based approach. In *Proceedings IEEE Visualization*, pages 155–162. Computer Society Press, 2002.

[13] Jihad El-Sana and Yi-Jen Chiang. External memory view-dependent simplification. In *Proceedings EUROGRAPHICS*, pages 139–150, 2000.

[14] Sarah F. Frisken and Ronald N. Perry. Simple and efficient traversal methods for quadtrees and octrees. *Journal of Graphics Tools*, 7(3):1–11, 2002.

[15] Enrico Gobbetti and Fabio Marton. Layered point clouds. In *Proceedings Symposium on Point-Based Graphics*, pages 113–120. Eurographics, 2004.

[16] Gevorg Grigoryan and Penny Rheingans. Probabilistic surfaces: Point based primitives to show surface uncertainty. In *Proceedings IEEE Visualization*, pages 147–154. Computer Society Press, 2002.

[17] J.P. Grossman and William J. Dally. Point sample rendering. In *Proceedings Eurographics Rendering Workshop*, pages 181–192. Eurographics, 1998.

[18] Matthias Hopf and Thomas Ertl. Hierarchical splatting of scattered data. In *Proceedings IEEE Visualization*, pages 433–440. Computer Society Press, 2003.

[19] Aravind Kalaiah and Amitabh Varshney. Modeling and rendering points with local geometry. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):30–42, January-March 2003.

[20] Leif Kobbelt and Mario Botsch. A survey of point-based techniques in computer graphics. *Computers & Graphics*, 28(6):801–814, 2004.

[21] Peter Lindstrom. Out-of-core simplification of large polygonal models. In *Proceedings SIGGRAPH*, pages 259–262. ACM SIGGRAPH, 2000.

[22] Peter Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. In *Proceedings Symposium on Interactive 3D Graphics*, pages 93–102. ACM SIGGRAPH, 2003.

[23] David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers, San Francisco, California, 2003.

[24] Renato Pajarola. Efficient level-of-details for point based rendering. In *Proceedings IASTED Invernational Conference on Computer Graphics and Imaging (CGIM)*, 2003.

[25] Renato Pajarola, Miguel Sainz, and Patrick Guidotti. Confetti: Object-space point blending and splatting. *IEEE Transactions on Visualization and Computer Graphics*, 10(5):598–608, September-October 2004.

[26] Mark Pauly, Markus Gross, and Leif P. Kobbelt. Efficient simplification of point-sampled surfaces. In *Proceedings IEEE Visualization*, pages 163–170. Computer Society Press, 2002.

[27] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings SIGGRAPH*, pages 335–342. ACM SIGGRAPH, 2000.

[28] Liu Ren, Hanspeter Pfister, and Matthias Zwicker. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. In *Proceedings EUROGRAPHICS*, pages 461–470, 2002. also in Computer Graphics Forum 21(3).

[29] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings SIGGRAPH*, pages 343–352. ACM SIGGRAPH, 2000.

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

[31] Miguel Sainz, Renato Pajarola, and Roberto Lario. Points reloaded: Point-based rendering revisited. In *Proceedings Symposium on Point-Based Graphics*, pages 121–128. Eurographics Association, 2004.

[32] Gokul Varadhan and Dinesh Manocha. Out-of-core rendering of massive geometric datasets. In *Proceedings IEEE Visualization*, pages 69–76. Computer Society Press, 2002.

[33] Jeffrey S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[34] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *Proceedings SIGGRAPH*, pages 371–378. ACM SIGGRAPH, 2001.