Made available by Hasselt University Library in https://documentserver.uhasselt.be

A data distribution strategy for parallel point-based rendering

Peer-reviewed author version

HUBO, Erik & BEKAERT, Philippe (2005) A data distribution strategy for parallel point-based rendering. In: Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG2005). p. 1-8..

Handle: http://hdl.handle.net/1942/6468

A Data Distribution Strategy for Parallel Point-Based Rendering

Erik Hubo
Expertise Center for Digital Media
Limburgs Universitair Centrum
Universitaire Campus
B-3590 Diepenbeek Belgium
erik.hubo@luc.ac.be

Philippe Bekaert
Expertise Center for Digital Media
Limburgs Universitair Centrum
Universitaire Campus
B-3590 Diepenbeek Belgium
philippe.bekaert@luc.ac.be

ABSTRACT

During the last couple of years, point sets have emerged as a new standard for the representation of largely detailed models. This is partly due to the fact that range scanning devices are becoming a fast and economical way to capture dense point clouds. Traditional rendering systems are impractical when a single polygonal primitive contributes less than a pixel during rendering. We present a data distribution strategy for parallel point-based rendering, using a cluster of PCs as target platform. We describe a data-structure and a system architecture, which allows for decoupling the point-data from the computational work. This strategy enables both a balanced workload as well as no full data replication on each node. We exploit frame-to-frame coherence to make our system scalable. The system renders high-resolution images from high complex data sets at interactive frame rates. To our knowledge parallel point-based rendering has not been investigated in the past. Our results indicate the feasibility of sort-first parallelization applied to point-based rendering.

Keywords

Cluster Computing, Parallel Rendering, Point-Based Rendering

1. INTRODUCTION

A recent trend in computer graphics is the shift towards sample-based rendering. Today's range sensing devices are capable of producing highly detailed and massive point clouds, which do not fit in the main memory of a single commodity PC. Point-based rendering can be more efficient than traditional rendering for these complex models if triangles occupy a small screen region. Processing many small triangles leads to bandwidth bottlenecks and excessive floating point and rasterization requirements [DeeM93]. Because of the absence of topology and relative positions, point-clouds are well suited for spatial subdivision and distribution between different PC's. One way of visualizing these enormous data sets is the use of expensive multiprocessor graphics servers with a huge main memory. A reasonable less expensive alternative of these dedicated graphics machines is a cluster of commodity PC's, linked by a high bandwidth network. The main challenge is to develop efficient parallel rendering algorithms that scale well within the processing, storage and communication characteristics of a PC cluster. Using this system architecture has many advantages: price-performance ratio, modularity, flexibility, storage capacity and scalability. Processing power, storage and memory capacity grow linearly with the number of PCs. A drawback to the traditional, tightly-integrated parallel computers is the fact that there is no fast access to a shared virtual memory space, and that the bandwidth and latencies of inter-processor communication are significantly higher. The challenge is to develop algorithms that evenly divide workload among PCs, do not introduce extra work due to parallelization and scale well as more PCs are added to the system.

In this paper we propose a data and work distribution scheme for parallel point-based rendering on a PC cluster.

This paper is organized as follows: first we discuss previous work in section 2. Next, we give a short system overview in section 3. In section 4 we present our implementation, data structures and system architecture. Finally, sections 5 and 6 discuss our results and conclusions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WSCG 2005 Conference proceedings ISBN 80-903100-7-9
WSCG'2005, January 31-February 4, 2005
Plzen, Czech Republic.
Copyright UNION Agency – Science Press

2. PREVIOUS WORK

Point-Based Rendering

During the last couple of years, there has been an increased interest of the computer graphics community in point based rendering techniques. Point-based rendering dates back as far as 1985, the year in which Levoy and Whitted [LeMa85] proposed the use of points to model and render 3D continuous surfaces. In 1989, Westover [WeLe89], introduced splatting for interactive volume rendering. Splatting algorithms handle volume data as a set of particles that absorb and emit light. Westover's basic splatting algorithm suffers from considerable artifacts due to inaccurate visibility determination when composing the splats from back to front. More recently, image-based rendering [McLe95] has become popular because the rendering time is proportional to the number of pixels (points) warped from the source to the output images. This contrasts with the scene-dependant time complexity for more traditional rendering techniques. Later on, the Lightfield [LeMa96] and Lumigraph [GoSt96] techniques were developed. These algorithms describe the radiance of a scene as a function of position and direction in a four-dimensional space, however, at the price of storage overhead.

One of the first point based rendering systems was QSplat [RuSz00]. In QSplat, a multi-resolution hierarchy, based on bounding spheres, is employed for the representation and progressive visualization of large models. The system is able to handle large meshes at constant frame rate. Pfister and Zwicker introduced surfels [PfHa00], short for surface elements. Surfels are a powerful paradigm for efficiently rendering complex geometric objects at interactive frame rates. Surfels can handle complex shapes; introduce low rendering cost and high image quality. Three orthogonal LDI's [ShJo98] are used to sample objects and image space filters are employed to achieve hole-free rendering. Later Zwicker et al. presented a framework for direct volume rendering [ZwMA01] using a splatting approach based on elliptical Gaussian kernels, superior to the footprints of Westover [WeLe89]. This results in high-quality anti-aliased rendering without excessive blurring. Botsch et al. proved that a pure software implementation could render up to 14 million Phong shaded samples per second by using a quantization of splat shapes [BoMa02]. However the models used to achieve these rendering times are not complex in terms of memory requirement. Their quantized hierarchical data representation is very compact with a memory consumption of less than 2 bits per point position.

Software-based point-based rendering algorithms have proven to be superior to polygon-based rendering algorithms for highly complex scenes. High quality results can be achieved but their rendering speed is limited. Recent algorithms use graphical hardware to overcome this problem. This idea was first introduced in [RuSz00]. In [CoLi02] the authors avoid using the z-buffer by sorting an octree from back to front each frame similar to McMillan [McLe95]. In [BoMa03] the authors provide high quality as well as efficient rendering based on a two-pass splatting technique with Gaussian filtering. Finally, in their most recent publication the authors propose to base the lighting of a splat on a linearly varying normal field associated with it, resulting in a visually high quality image [BoMa04]. Dachsbacher et al. [DaCa03] present a hierarchical LOD structure that is suitable for GPU implementation. They can process 50M low quality points per second

A main drawback of all the GPU algorithms is that they only perform well on rather simple models with a low screen resolution. This is due to the fact that, although extremely fast, a GPU's on-board memory is currently rather limited in terms of data storage. To overcome this limitation we use a PC cluster to speed up the rendering. Since PC clusters have a scalable memory capacity, they are well suited for the interactive rendering of high-resolution images of complex models.

A short overview of parallel rendering is presented next.

Parallel rendering

Parallel rendering systems have long been used for ray tracing [WaIn01], radiosity and global illumination [FuTh96, ZaDa95, ReEr98]. These systems can often be classified by the stage in the graphics pipeline in which the primitives are partitioned: sort-first, sort-middle or sort-last [MoSt94]. In sort-first systems, screen space is partitioned in non-overlapping 2D tiles, each of which is rendered independently. The final image is obtained by composing all 2D tiles. The main advantage of this method is the low communication cost. The efficiency of sort-first algorithms is limited by redundant rendering due to overlapping tiles [SaRu01]. In general, since the overlap factors grow with increasing numbers of processors, the scalability of sort first systems is limited [MuCa95]. Sort-middle, the most straightforward approach, is commonly used in traditional systems. Primitives are redistributed in the middle of the rendering pipeline, between geometry processing and rasterization. This approach is not well suited for a cluster of PC's due to its high communication requirements. Finally sort-last methods defer sorting until the end of the rendering pipeline. The main advantage of sort-last is its scalability [MoSt94].

In the last few years, there has been a growing interest in PC clusters for interactive rendering tasks. Humphreys and Hanrahan presented a sort-first system designed for 3D graphics called WireGL [HuGr99, HuGr00]. WireGL was used to achieve

scalable display size with minimal impact to the application's performance. Unlike sort-middle, sort-first can use retained-mode scene graphs to avoid most data transfers for graphics primitives between processors [MuCa95]. In [SaRu00] a hybrid sort-first sort-last approach for parallel polygon rendering is presented. A specific algorithm for dynamic, view-dependent and coordinated partitioning is used of both the 3D model and the 2D image, which has positive results in terms of both performance and scalability.
Continual growth in typical dataset size and network bandwidth has made stream-based analysis a hot topic for remotely stored 3D models [RuSz01]. Streams are appropriate computational primitives, because large amounts of data arrive continuously, and it is impractical or unnecessary to retain the entire dataset. Chromium [HuGr02] is another a stream-processing framework based on WireGL. Its stream filters can be arranged to create sort-first and sort-last parallel graphics architectures.

Since we are interested in high-resolution images, we prefer a PC cluster method to the recently popular GPU methods because of its scalable memory capacity. High-resolution images require complex models with many point samples, which cannot be accommodated by the memory of the graphical hardware. We believe our sort-first parallelization is scalable because the overlap factor is negligible in point-based rendering. To the authors' knowledge parallel point-based rendering has not been investigated in the past.

3. SYSTEM OVERVIEW

Our system operates in two stages:

Preprocessing Stage: The first stage serves as an offline preprocessing stage and is only performed once per 3D model. Details are provided in section 4.1. The input for the first stage is a point- cloud. The system creates a multi-resolution hierarchical spatial subdivision structure, optimized for fast data traversal.

Rendering stage: The second stage is the render stage. We use four types of processes in our system architecture to decouple the data from the computation in order to achieve an optimal load balance. We briefly describe these processes of the rendering pipeline below (Details are provided in section 4.2 to 4.5):

Display process: This process executes the first and last stage of the rendering pipeline. In the first stage, the display process divides the view frustum into a set of smaller mini view frusta, according to a box of interest, and sends them together with camera data to the data traverse processes. After computation in the final stage, the display process receives the images corresponding to these mini frusta and loads them into the framebuffer for display.(see figure 1 (a)).

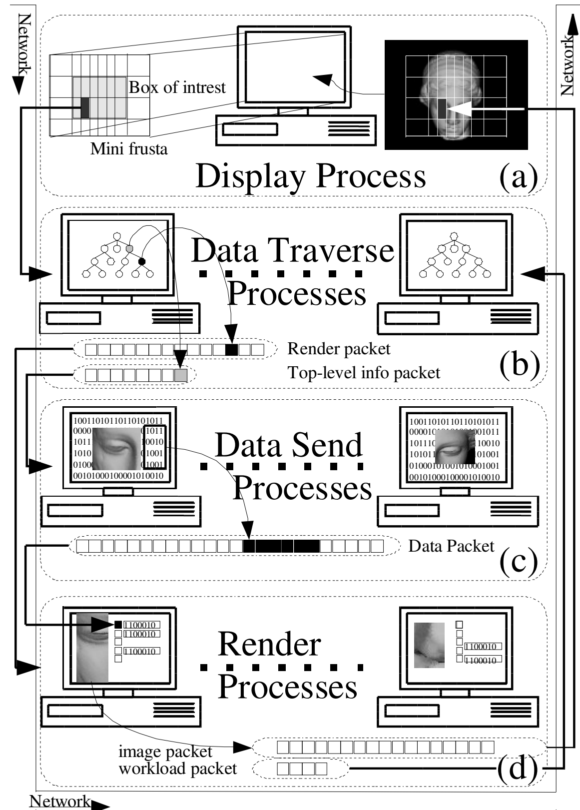


Figure 1: System overview of the rendering pipeline: (a) Display process: Frustum subdivision according to a box of interest and display. (b) Data traverse processes: traversing data and gathering render information. (c) Data send processes: sending point-data. (d) Render processes: Caching and rendering the incoming data and sending the rendered images back to the display node.

Data traverse process: A data traverse process requests a mini frustum from the display process. While traversing the octree data structure, the data traverse process clips the octree cells against the mini frustum, and decides which octree cells are suitable for rendering. For each mini frustum the data traverse process maintains, together with the list of useful octree cells, a list of used top-level octree cells. These are hierarchically higher octree cells (see figure 3). Depending on the workload and the available data on the render nodes (see section 4.5), the data traverse process can correctly determine the render node the data should be sent to. (see figure 1(b)).

Data send process: The data traverse processes inform the data send processes what point-data should be sent to which render node (see section 4.5). (see figure 1(c)).

Render process: Render processes receive packets from data send processes (data packets) and from data traverse processes (render packets). Data packets contain point-data of a top-level octree cell. Render packets contain pointers to the data that has to be

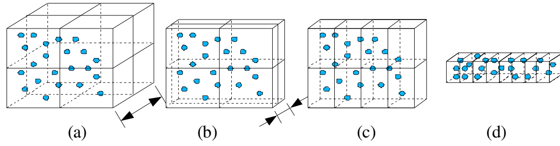


Figure 2: the octree cell (a) shrinks to the bounding-box of its points: (b). Subdividing a small edge (depth edge in (b)) should be avoided. Therefore (b) could be subdivided like (c). (d) Good solution for this case.

rendered, camera and mini frustum data. Received data packets are temporarily stored on the render node (see section 4.5). A render node creates one image per received render packet, assuming all necessary data packets are available. This image is sent back to the display process. (see figure 1 (d)).

4. IMPLEMENTATION

In this section we describe the implementation of our distributed point-based-rendering system in detail, and comment on the applied data structures and algorithms.

Preprocessing

The preprocessing stage is the first stage in the algorithm and has to be executed only once for any given input point cloud. Like other point-based rendering algorithms [RuSz00, BoMa02], an octree based hierarchical spatial subdivision structure is created from an input point cloud. The advantages of this data structure are: (1) fast data traversal: frustum and backface-culling, optimal succession of octree cells cache coherence [ChTr99] (2) immediate access to all data in an octree cell (for data sending) (3) multi-resolution. If no normals or splat sizes per 3D position are included in the point cloud, these data can be simply derived from sample neighborhoods.

4.1.1 Octree

We construct the octree data-structure using a two-step procedure. First, we create an ordinary axis-aligned octree. Since we are working with large datasets, special care has to be taken to limit the octree recursion, which could adversely affect the algorithms efficacy. Therefore we assume local neighborhoods to be planar. While subdividing the octree, the algorithm resizes each octree cell to the bounding box of points located in this cell (see figure 2 (a) and (b)). Since this changes the proportions of the octree cells, these cells could be subdivided amongst there biggest edge(s) (see figure 2 (c)(d)) for an optimal spatial division. The leaf octree cells contain the actual point-data.

In the second step the heavy loaded octree is rewritten to a fast, compact and memory-coherent octree. Initially, we split the point-data from the octree. The algorithm recursively creates the *point-array*. This array is sorted in such a way that every octree cell has a start index and a size to access its point-data in this *point-array* (see figure 3). This is useful when we need fast data-access to a non-leaf octree cell. Besides a start index and size to its data, each octree cell contains location, normal, normal cone and bounding box information. Each octree cell has some structural information: a level (section 4.1.2), an index to its sibling, and an index to its top-level octree cell (see figure 3). All the data of the octree cell is aligned in 64 bytes for cache-performance reasons. If an octree cell has no siblings it has a recursive index to its parent's sibling (see figure 3: octree cell 10's sibling). A top-level octree cell is a uniform parent at a low depth in the octree: it shares the same point-data as any octree cell beneath it. Each octree cell has an index to the top-level octree cell that contains its data (see figure 3). To align the data structure and avoid cache trashing [ChTr99] we write the octree down to an array, the *octree-cell-array*, by traversing the octree in depth-first order (the same order as the data traverse processes use (see figure 3)) This way we do not need to save a pointer to the first child of an octree cell.

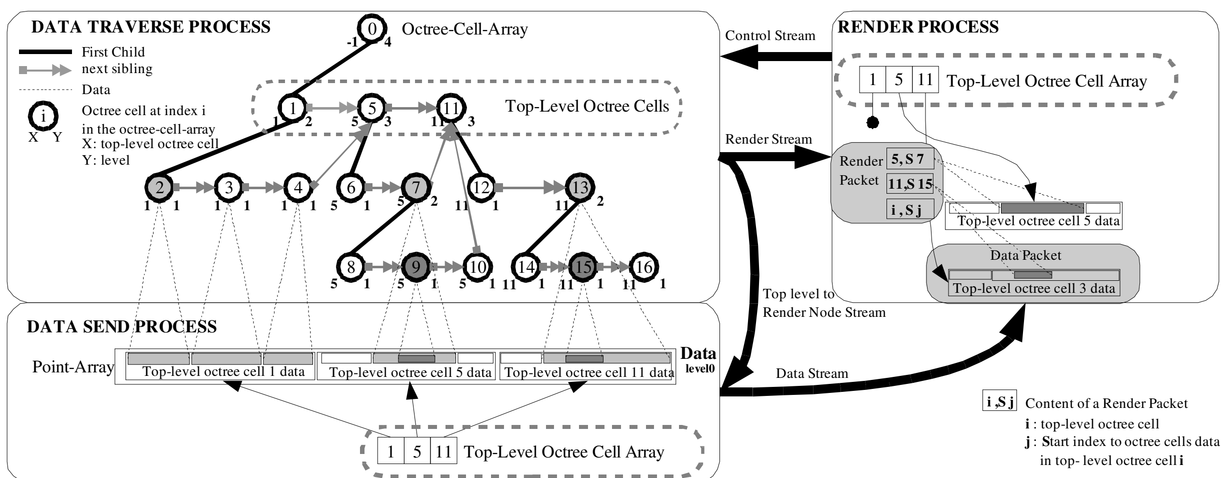


Figure 3: The octree data structure: Data Traverse Process: octree written down to an array, all information available except the point-data. Data Send Process: Top-level octree cell array pointing to point-array. Render Process: Top- level octree cell array pointing to received data packets. Render packets show what has to be rendered.

4.1.2 Multi Resolution

It is not necessary to use the full point-data for a model far from the camera. It is better to use a compact version of the data to save processing and network resources. Other algorithms, e.g. [RuSz00], use the information in their spacial subdivision scheme to create a multi-resolution model. Since we decouple the data structure from the point-data, we cannot introduce multi-resolution point-data in the data-structure. Therefore *level-splats* are introduced. As we mentioned in the previous section, every octree cell has a level (see figure 3). Data-points have level zero, leaf octree cells have level one, and the levels of all other octree cells is one more than the maximum level of their children (see figure 3). To create level(n) splats we build for each level(n) octree cell a spatial subdivision data-structure on its level(n-1) splats. We use this data structure together with a covariance analysis [PaMa02] (Mahalanobis distance [JoIT]) to cluster level(n-1) splats to level(n) splats.

Display process

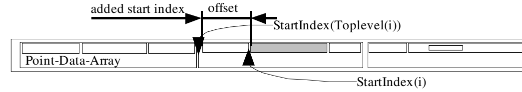
The system contains only one display process, which provides the user-interaction. The display process dynamically divides the view frustum into mini view frusta. This is a sort first approach [MoSt94]. The dimensions of these mini frusta are computed considering a box of interest. Typically this box is the bounding box of the point-data. The display process sends these mini frusta together with camera data and a timestamp to data traverse processes that reported to be idle. The display process keeps a queue of incoming images and sequentially displays these.

Data traverse process

A data traverse process only loads the octree-cell-array (see section 4.1.1) into its main memory. This implies that the data traverse processes can work on the entire data set without loading the massive point-data. This way the computational work can be decoupled from the data, resulting in a well-balanced workload. Each idle data traversing process asks the display process a new mini frustum and creates a render packet associated with it. This render packet is filled during the traversal of the octree as described below:

```
TraverseOctreeCellArray(){
   int index = 0;
   do
      if(whole array[index] in mini frustum)
         AddToPacket(index); index = siblingindex
      else if( part of array[index]  in mini frustum)
         if(array[index] benefit of  subdivision is high)
               index++
         else
               AddToPacket(index); index = siblingindex
      else if(array[index] out mini frustum)
         index=siblingindex
   while(index exists)  }
```

Where *array* is the *octree-cell-array*, *index* is the position in this array of the octree cell that we are using and *siblingIndex* is the position of the sibling of this octree cell in the *octree-cell-array*. This function exploits the structure of the *octree-cell-array* and avoids cache trashing [ChTr99]. Furthermore it uses frustum culling and decides whether the benefit of examining the children of the octree cell is sufficient. The AddToPacket function works on the octree cell at position *index* in the *octree-cell-array*. First we try to backface cull the octree cell, considering its normal and normal cone. If the top-level octree cell of the octree cell does not exist, we are too high in the octree and need to examine the children of the octree cell. The algorithm decides which data resolution it should use depending on the screen resolution, the octree cells distance to the camera and the available data resolutions for this octree cell. The size and the start index of the octree cells data are added to the render packet. The added start index is the offset from the octree cells data to the top-level octree cells data (see figure 4).



Added Start index = StartIndex(i) - StartIndex(toplevel(i)).

Where i is an octree cell.

Figure 4: Added start index is the offset from the octree cells data to the top-level octree cells data.

The indices of the used top-level octree cells are also added to the render packet. When the octree traversal is finished, the render packet is ready. Every data traverse process has information concerning the current workload and the available data on each render node (see section 4.5). The render node with the smallest cost is chosen to receive and render the render packet. The cost is computed as described next:

Cost(i) = Render Cost(i)+Network Cost(i)

Render Cost(i) = workload on render process(i)*Ts

Network Cost(i) = unavailable data on render process(i) * Tn

Where i is a render node, Ts is the time to render one splat and Tn is the inverse network speed. Finally the data traverse process informs all data send processes what unavailable point-data they need to send to the chosen render node.

Data send process

A data send process loads the *point-array,* or a part of it, grouped per top-level octree cell in its main memory (see figure 3).

Data send processes receive their instructions from the data traverse processes; they inform the data send processes to which render node which top-level octree cells data should be sent (see figure 3). Data send processes always send the entire point-data of a top-level octree cell.

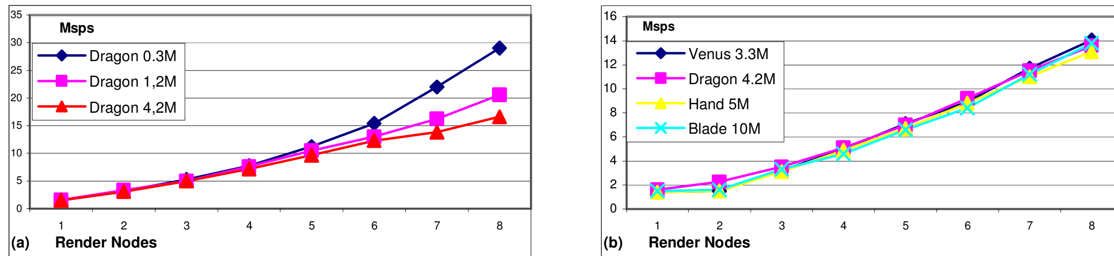


Figure 5:(a)(b) We averagely splat 1.5 Million Splats per Second per render node. If the model grows in complexity the splat rate could drop a little because the cost of traversing the octree increases.

Render process

In [MoSt94], the authors state that a sort first approach is only scalable if the frame-to-frame coherence is exploited. Therefore, we introduce top-level octree cells. These are regular octree cells at a low depth in the octree (depth three, four or five depending on the size of the model). Combined, all top-level octree cells mutually exclusive enclose the entire *point-array* (see figure 3). When using top-level octree cells we avoid both redundant data in the cache of our render processes and high network traffic. Furthermore, we exploit the frame-to-frame coherence, by sending more data than directly needed.

A render node is a separate workstation running four render processes that share the same memory place. A render node receives two kinds of data streams, one from the data traverse processes and one from the data send processes. Initially, each render node contains an empty array with all top-level octree cells. The point-data in this array is filled each time point-data of a top-level octree cell is received from a data send process. Render packets, sent by the data traverse processes, contain pointers to the point-data of the octree cells that lie in the mini frustum. Each pointer is an offset in the point-data of the top-level octree cell where the pointers octree cell belongs to (see figure 3 and 4). Render packets also indicate which top-level octree cells point-data should be available to render this packet. If all requested point-data is available, an idle render process will render the packet. As long as the requested data is not available, the packet will be queued. To avoid running out of memory, a least recently used caching scheme is applied. The least recently used point-data of a top-level octree cell will be deleted after a time-out period has expired. All data traverse processes will be informed about this, so they can recompute the cost of sending data to that render node. For the same reason, render nodes inform the data processes about their current workload, this is the amount of points they still need to render. The rendered image is sent back to the display process for composition and display.

In our current framework we use a simplified EWA [ZwMA01] splatting algorithm that could be easily replaced by a more advanced splatting algorithm if required.

5. RESULTS

The PC cluster used for our experiments consist of 9 workstations. Each node has two 2.4 Ghz Intel Pentium IV Xeon processors, 2 GB DDR Ram, and is running Suse Linux 9.1. The nodes communicate with the LAM MPI implementation through a gigabit network. Since we are using a purely software based implementation, we exploit the computational power of each workstation and run several processes simultaneously. In our test setup the system runs as many Data Traverse as Render Processes (please note that there is not a one-to-one mapping between these processes.)

Scalability

5.1.1 Model Complexity

We first consider the scalability of our system with regards to the model complexity. We have two test cases: (1) three dragon point sets with 0.3M, 1,2M and 4,2M points. (2) Different models with different complexities: Dragon 4,2M points, Turbine Blade 10M points, Hand 5M points and Venus 3M points.

5.1.1.1 Splats Per Second

Our experiments showed that if we use only one render node, we are able to splat an average of 1.5

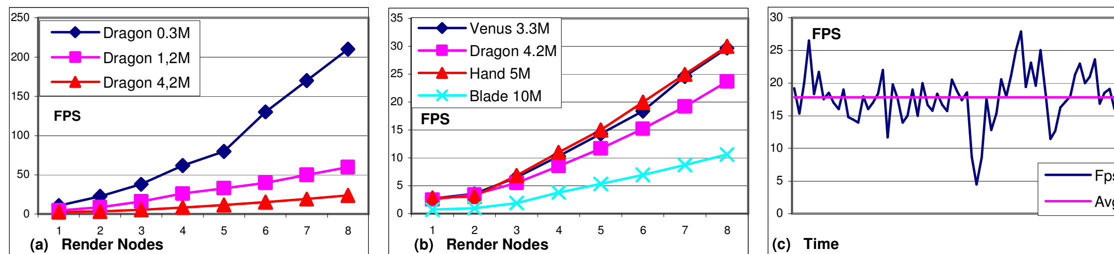


Figure 6:(a)(b) If the model grows in complexity more point are needed each frame. Since the splat rate is rather constant, the frame rate will drop. For non-complex models we could render up to 210 fps while very complex models still result in 11 fps.(c) the frame rate is rather constant

Million Splats per Second, if all necessary data is available on the render node. Figure 5 shows the scalability of the splats per second. If the model grows in complexity, the global splat rate could drop a little because the cost of traversing the octree increases (the difference between figure 5 (a) Dragon 0.3M and (b) Blade 10M).

5.1.1.2 Frames per Second

If the model grows in complexity, more points need to be rendered each frame. Since the splat rate is more or less constant (see figure 5), the frame rate will drop for these complex scenes (see figure 6 (a)). However, it is not necessary to render more points than dictated by the screen resolution. This means the frame rate does not entirely depend on the complexity of the model, it also depends on the screen resolution and the available model resolutions. We could speed up the frame rate by choosing the optimal model resolution for each octree cell, depending on its distance to the camera and the screen resolution (see figure 6(b)). This results in a scalable frame rate.

Figure 6 (c) shows us that the frame rate is rather constant. If the frame rate drops, point-data packets are sent.

5.1.2 High Resolution

A small part of the computational power is spent on sending images to the display process that loads them to the graphics board. This implies that the performance of our system is not very sensitive to the screen resolution, if the number of splats stays constant. As we can see on figure 7(a) the frame rate only drops if the resolution becomes too high. This is a result of the high communication costs associated with sending high-resolution images. However, if the number of splats increases with the resolution, as we described section 5.1.1, the frame rate will drop faster (see figure 7(b)), because more points need to be rendered. However, the quality of these images will be higher.

Load Balance

Each render node has a cost to render a given render packet. The correct choice of the render node with the smallest cost (see section 4.3 data traverse node) is vital for good load balancing. In figure 7(c) the workload for 8 render nodes is depicted, during the rendering of the Turbine Blade point set (10M points). When our process starts, the workload is low because many point-data packets are sent to the render nodes. Figure 7(c) clearly indicates that our cost function and system architecture is well chosen, because all render nodes are almost equally loaded and the global workload does not drop too much. When the workload drops, point-data packets are sent.

6. CONCLUSION

This paper presents a scalable data distribution strategy for parallel point-based rendering on a PC cluster architecture. Since the used data-structure and the algorithm's architecture decouple the data from the computational work, the system achieves a well balanced workload and each data traverse process can work on the entire data without a full replication of the data. The algorithm dynamically partitions the screen into smaller mini frusta (a sort-first approach). Our technique exploits the sort-first properties of the algorithm, by sending more data than is directly needed. Large data sets at high screen resolution can be rendered at interactive frame rates. Point-Based rendering is well suited for a sort-first parallel rendering approach because the overlap factor is negligible.

Topics for further study include faster software point-splatting algorithms with higher quality, using low-level processor instructions. Also, combining clustered CPU and GPU rendering might be an interesting research venue.

7. ACKNOWLEDGEMENTS

We would like to thank everybody who helped us with this publication and the Stanford Computer Graphics Laboratory for sharing the models used in our experiments.

8. REFERENCES

[BoMa02] M. Botsch, A. Wiratanaya L. Kobbelt, Efficient high quality rendering of point sampled geometry, 13th Eurographics workshop on Rendering, pp 53-64, 2002.

[BoMa03] M.Botsch, L.Kobbelt,High-Quality Point-Based

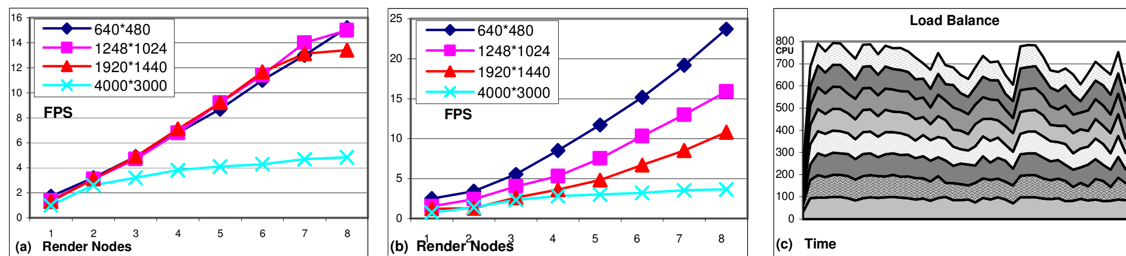


Figure 7:(a) Tests are done with the dragon 4M point set. The system is, if the number of splats stays constant, only sensitive to the screen resolution if the overhead of sending the images back to the display node is too high (b) the frame rate drops faster because the number of splats increases with the resolution (c) the workload for 8 render nodes, during the rendering of the Blade point set (10M points).

Rendering on Modern GPUs, 11th Pacific Conference on Computer Graphics and Applications, pp 335,2003.

[BoMa04] M.Botsch,M.Spernat,L.Kobbelt,Phong Splatting, pp 25-32, Symp.on Point-Based Graphics,2004.

[CaEd74] E. E. Catmull, A subdivision algorithm for computer display of curved surfaces.1974.

[ChTr99] T.M. Chilimbi, M. D. Hill, J. R. Larus, Cache-Conscious Structure Layout, Programming language design and Implementation SIGPLAN99, pp 1 –12, 1999.

[CoLi02] L. Coconu, H Hege, Hardware-accelerated point-based rendering of complex scenes, 13th Eurographics workshop on Rendering, pp 43- 52,2002.

[DaCa03] C. Dachsbacher, C Vogelgsang and Marc Stamminger, Sequential point trees, Trans. Graph.,pp 657-662, 2003.

[DeeM93] Data Complexity for virtual reality: where do all the triangles go?, IEEE Virtual Reality Annual International Symposium, pp 357-363, 1993

[FuTh96] T. A. Funkhouser, Coarse-Grained Parallelism for Hierarchical Radiosity Using Group Iterative Methods, Computer Graphics,pp 343-352, 1996.

[GoSt96] S.J.Gortler, R. Grzeszczuk,R. Szeliski,M. F. Cohen, The Lumigraph,SIGGRAPH96, pp 253–262, 1996

[HuGr99]G. Humphreys, P. Hanrahan, A distributed graphics system for large tiled displays, Proceedings of the conference on Visualization '99,pp 215-224,1999.

[HuGr00]G. Humphreys, I. Buck, M. Eldridge,P. Hanrahan, Distributed rendering for scalable displays, ACM/IEEE conference on upercomputing,pp.30,2000.

[HuGr02]G. Humphreys, M. Houston, R. Ng,R Frank, S. Ahern,P. D. Kirchner,J. T. Klosowski, Chromium: a stream-processing framework for interactive rendering on clusters, Computer graphics and interactive techniques,pp 693-702 ,2002.

[JoIT]I.T. Jolliffe, Springers Series in Statistics, Principal Component Analyse, second edition, pp 92- 93. ISBN 0-387-95442-2

[LeMa85] M. Levoy, T. Whitted, The use of points as display primitive. Tech. Rep. TR 85-022, University of North Carolina at Chapel Hill.

[LeMa96] M. Levoy, P. Hanrahan, Light Field Rendering,SIGGRAPH96,pp 31 – 42,1996

[McLe95] L. McMillan, G. Bishop, Plenoptic Modeling: An Image-Based Rendering System, pp 39-46, 1995.

[MoSt94] S. Molnar,M. Cox, D. Ellsworth,H. Fuchs, A Sorting Classification of Parallel Rendering, IEEE Computer Graphics and Algorithms, p23-32, 1994.

[MuCa95] C. Mueller, The sort-first rendering architecture for high-performance graphics, symposium on Interactive 3D graphics, pp 75 - end, 1995.

[PaMa02] M. Pauly, M. Gross, L.P. Kobbelt, Efficient simplification of point-sampled surfaces, IEEE Visualization pp 136- 170,2002.

[PfHa00] H. Pfister, M. Zwicker,J.v. Baar, M. Gross, Surfels: Surface Elements as Rendering Primitives, SIGGRAPH00, pp 335-342,2000

[ReEr98] E. Reinhard, A. Chalmers,F. W. Jansen, Overview of Parallel Photo-realistic Graphics, nr CS-EXT-1998-147, 1998.

[RuSz00] S. Rusinkiewicz, M. Levoy, QSplat: A Multiresolution Point Rendering System for Large Meshes, pp 343-352, Siggraph00, 2000.

[RuSz01] S. Rusinkiewicz, M. Levoy,Streaming QSplat: a viewer for networked visualization of large, dense models, symposium on Interactive 3D graphics,pp 63-68, 2001.

[SaRu00] R. Samanta,T. Funkhouser,K. Li, J. Pal Singh, Hybrid sort-first and sort-last parallel rendering with a cluster of PCs, Eurographics workshop on Graphics hardware, pp 97-108, 2000.

[SaRu01] R. Samanta,T. Funkhouser,K., Parallel Rendering with K-way Replication, IEEE 2001 symposium on parallel and large-data visualization and graphics, pp 75 –84, 2001

[ShJo98] J.Shade, S. Gortler,L. He R. Szeliski, Layered depth images, Computer graphics and interactive techniques, pp 231 –242, 1998.

[WaIn01] I. Wald, P. Slusallek, C. Benthin, Interactive Distributed Ray Tracing of Highly Complex Models, EUROGRAPHICS, Workshop on Rendering, pp 277-288, 2001,

[WeLe89] L. Westover, Interactive volume rendering, Chapel Hill workshop on Volume visualization pp 9-16, 1989.

[ZaDa95] D. Zareski, B.Wade, P. Hubbard,P. Shirley, Efficient Parallel Global Illumination Using Density Estimation,IEEE/ACM 1995 Parallel Rendering Symposium (PRS '95),pp 47- 54, 1995.

[ZwMA01] M. Zwicker, H. Pfister, J.v.Baar, M.Gross, Ewa volume splatting, IEEE Visualization 2001, pp 29-36, 2001.

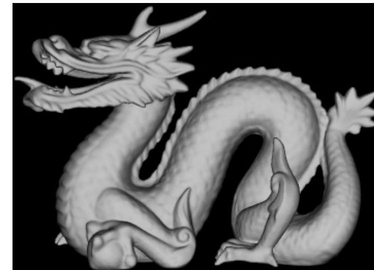


Figure 8: test model: Dragon 4M points

Figure9: test model: Hand 5M points

Figure 10: test model: Turbine Blade 10M points