IADIS International Journal on Computer Science and Information Systems
Vol. 11, No. 2, pp. 146-158
ISSN: 1646-3692

SMOOTH VISUALIZATION OF LARGE POINT CLOUDS

Jörg Futterlieb, Christian Teutsch and Dirk Berndt. *Fraunhofer Institute for Factory Operation and Automation IFF, Magdeburg, Germany*

ABSTRACT

We present a novel approach for processing and rendering large point cloud data from 3D scanners on a standard computer system. Our technique handles a data size that typically exceeds the RAM for processing and the VRAM for rendering the data. We achieve interactive framerates and a smooth visualization by enhancing existing large data visualization techniques, like Level of Detail and deferred rendering approaches and we utilize the new capabilities of modern GPU hardware. This enables the exploration of large and unstructured 3d scan data sets whose size is only limited by the hard disk memory available. We verify our approach at the example of different data sets with up to one billion 3D points.

KEYWORDS

3D Point clouds, Out-of-core, Level of Detail, Deferred rendering, Interactive visualization

Figure 1. From left to right: Zoom into a 3D point cloud of earth data with 1 billion points (file size is 42.2 GB). Our proposed technique provides a smooth exploration on a standard computer system

1. INTRODUCTION

Non-contact optical methods of 3D digitization have replaced conventional methods in many applications. Regardless of the varying 3D measuring principles - many millions of measured 3D data are generated within a few seconds. This data delivers a description of a digitized component's surface topography. 3D scanning has a wide distribution in different industrial applications, e.g. in quality assurance or reverse engineering. Even whole environments from buildings to large industrial plants are digitized which results in a large amount of 3D points that need to be processed and visualized.

The rapidly developing market of 3D sensors provides high-speed light-section scanners for industrial measurement applications that produce 3D scanlines of 1000 points with a frequency of 1000 Hz. So it takes just a second to generate a million 3D points. Even consumer products like a Microsoft Kinect generate more than 300.000 points per scan. Especially terrestrial laser scanners (TLS) and time of flight (TOF) cameras show a rapid development with regards to acquisition speed and scan data resolution.

Very often several scans from different places or viewing directions are aligned and merged together to produce a connected and comprehensive 3D representation. Finally we have to deal with hundreds of millions and up to a few billions of 3D points with many gigabytes of data which makes the fast processing and rendering of the 3D data a challenging task. In this work we are facing this problem with efficient methods for 3D data management and processing while keeping a special focus on large 3D data visualization approaches based on deferred rendering methods using standard computer systems.

Large 3D data visualization is an active research area with several strategies and methods presented in the last years. This paper is an extended version of Futterlieb et al. (2016) and structured as follows: We discuss previous and related work in section 2. In section 3 we present our approach for the interactive visualization of large 3D point clouds, including the used hierarchical data structures, enhanced Level of Detail (LOD) algorithms and the deferred rendering pipeline. Section 4 discusses experimental results achieved with our implementation. Section 5 concludes and summarizes this work.

2. RELATED WORK

The interactive visualization of large 3D point clouds has three major topics that we need to consider. At first structuring the raw 3D point cloud data is discussed because it needs to balance efficient data storage and fast data access for the 3D search and query algorithms. In order to provide a real-time interaction Level of Detail techniques for reducing the point cloud complexity are considered as the second task. Lastly, we consider approaches for the smooth and appealing visualization with latest deferred rendering methods.

Structuring the raw 3D point data is an integral component of our visualization framework, because the selection of visible points and the access to certain detail levels requires efficient spatial neighborhood queries. For the visualization of large volume data variants of Octrees are typically used (Wenzel, 2014; Wimmer & Scheiblauer, 2006), because they subdivide the entire space into regularly sized cells and voxels which can be addressed and accessed easily. They have proven to perform well for real-time 3D collision detection (Weller et al., 2013). Due to the regular subdivision scheme the efficiency of Octrees strongly depends on the

distribution of data within the volume considered. 3D point clouds from optical (laser) scanners are representing an object's surface but not a volume, and additionally, these point clouds are often composed of partly overlaying scans which results in an inconstant point density. Thus, more flexible spatial data structures like AABB-trees and the variants of kd-trees (Richter et al., 2015; Goswami et al., 2012) are more suitable for our task. Kd-trees already show a high performance for 3D point clouds, e.g. for point cloud registration and pre-processing like smoothing, thinning and segmentation (Teutsch, 2007). Data storage and stream-wise processing of the huge point cloud is often done by well-known memory mapping techniques (Rusinkiewicz & Levoy, 2000; Dementiev et al., 2005). Additionally, there are Big Data storage schemes, where the data is stored and processed on a distributed filesystem (Boehm, 2014).

The second task is the application of Level of Detail (LOD) techniques. These methods generate down-sampled representations of the original (huge) data set in order to compute a resolution which depends on the distance between the virtual camera and the 3d data. The LOD may be discrete or hierarchical. The discrete methods provide various models in different resolutions that represent the same object, which are computed in advance. They are typically used for the LOD of polygonal meshes (Ribelles et al., 2010). On the other hand hierarchical methods encode the detail levels in a tree structure, where the tree layers provide representative points for specific volume clusters (Richter et al., 2015; Goswami et al., 2012).

The third task, the rendering part, is done by using pure point primitives or subsets of points which are approximated by another primitive e.g. the QSplat approach (Rusinkiewicz & Levoy, 2000). Botsch et al. (2005) gives an overview on point-based rendering techniques. They also discussing a GPU-accelerated approach with Deferred Shading, a render pipeline we further extend within this paper. Classic culling techniques like Frustum Culling, Backface Culling (Rusinkiewicz & Levoy, 2000), which are mostly known from polygonal meshes, are also appropriate for point clouds. Since most approaches focus on rendering just one data set, we also describe a method to handle and visualize many data sets simultaneously. Therefore, we introduce a scheduling mechanism which always ensures framerates required for interactive visualization. The overall performance and the interactive visualization are driven by a deferred rendering approach (Ferko, 2012), which reduces the amount of data to be rendered in one rendering call in contrast to classical fixed-pipeline techniques.

3. RENDERING LARGE DATASETS

The choice of our data structure is determined by a trade-off between two objectives. On the one hand, we would like to use bounding volumes for the LOD approach that need only a few bytes to store them, and that enable fast visibility tests and distance computations. On the other hand, we want the bounding volumes fit the corresponding 3D point cloud data tightly. Thus, we decided to use a variant of a kd-tree (comparable to the approach Richter et al., 2015) as a common spatial 3D structure both for processing and visualization of the point clouds. In each level of the tree hierarchy the splitting planes provide the required bounding boxes. We benefit from the fact that the kd-tree is always balanced and that it encodes the required detail levels in its hierarchy with an even, regular distribution. Although the build time for a kd-tree is higher than e.g. for an octree, neighborhood search queries are processed in less time on average (Wenzel, 2014; Teutsch, 2007). The data storage and access to the

large amount of data is done via file mapping methods provided by the operation system. Our kd-tree build routine takes care that neighboring points in deep hierarchies are stored close to each other. We also render pure point primitives instead of approximating subsets by some primitives (like in the QSplat approach from Rusinkiewicz & Levoy, 2000). This is important for the later data analysis steps, where we need to ensure that all visible data is real scan data and not an approximation.

In the following we describe the data structures and algorithms for handling and rendering the point cloud data. The concept is divided into three sections, the Level of Detail techniques to select visible points from the data set, the pipeline for rendering the point data and lastly the scheduling of multiple point clouds.

3.1 Level of Detail Techniques

In order to guarantee a smooth and interactive visualization we need to adaptively select a subset of the whole point cloud for two reasons. At first the entire data set does not fit into the main memory (RAM) or the graphics memory (VRAM). Thus we need to load from the hard disk only that amount of points that is currently visible to the user. Secondly we further reduce the amount of visible points considering that it cannot be larger than the total number of pixel of the screen (~ 2M for a HD screen with 1920x1080 pixels). The less data we load from the filesystem the faster the visualization will be since we want to achieve a real-time rendering and a smooth interaction with at least 60 frames per second.

Our rendering concept starts with a low resolution overview point cloud (2M points), which is statically stored in the VRAM and always ensures that some representative data is visible between detail level switches. The transition between detail levels itself is smoothed by applying the deferred rendering approaches in the later steps. Depending on the distance of the virtual camera to the 3d data and the size of the viewing volume we dynamically extract a number of points from the detail levels of the kd-tree.

3.1.1 Extracting Levels of Detail

Since kd-trees recursively split the 3D volume into clusters of nearly the same point count, we take the corresponding and required detail levels directly from the tree hierarchy. The detail level count equals the tree depth, which is the logarithm of the point count. The amount of overview points is controlled to guarantee a framerate of at least 60 Hz. Limitation of the point count is primary induced by the fill rate of the graphic card and less by the graphic cards memory size. To compute an adequate representative overview of the large data we use a tree level computed by the following formula:

$$Treelevel_{Overview} = \log_2(\mathrm{N}_{\mathrm{OverviewPoints}})$$

We define the number of overview points as maximum value. If the computed tree level has a non-zero decimal place, the tree layer can be rounded to the next lower integer value to maintain the point count limit. We use the next higher integer value and only extract the specific number of overview points from the tree, whereby the points are chosen coincidental (shuffling with a fixed seed). Figure 2 shows various detail levels for the overview point cloud.

Figure 2. Overview point cloud with various point counts, starting with 125000 points (left) and doubled until 2 million points (right). The kd-tree ensures a well suited representation of the scanned volume at each point count limit

It is possible to build one kd-tree from multiple data sets or each data set manages an own kd-tree. If multiple kd-trees are used, the overview point cloud is partitioned into sections. Each section size depends on the portion of the data set size in relation to all data sets. We compute the tree level for each point data set by scaling its point number ($N_{\text{Data set points}}$) with the number of total points of all data sets ($N_{\text{Total Points}}$):

$$Treelevel_{Data\,set} = \log_2\left(\frac{N_{\text{Data set points}}}{N_{\text{Total Points}}} * N_{\text{OverviewPoints}}\right)$$

The overview point cloud is instantly rendered and provides a global orientation for the user at any time. To present the detailed and dense point data from specific viewpoints, the visible points are selected from all data sets. This finally can lead to slow I/O-Operations of the hard disk. Therefore, we query the view-specific detailed points in separate threads provided by a thread pool (Garg & Sharapov, 2001). This decouples the rendering and the detail extraction to preserve always an interactive and smooth visualization.

3.1.2 Extracting view-specific Details

At first we apply a view frustum culling step by using the six frustum planes and the axis-aligned bounding box of the point cloud to compute the visibility (Sunar et al., 2006). If the point cloud is visible and the overview point cloud does not contain all points, we step into the kd-tree to query more detailed data.

We partition the frustum into successive axis-aligned bounding boxes for a fast point query from the kd-tree. A test for a point inside an axis-aligned bounding box is much cheaper than testing against six frustum planes, but viewing aslope along coordinate axes and especially in perspective projection the bounding box of the frustum also covers a large non-visible volume. We prevent that by dividing into smaller and disjoint axis-aligned boxes. Furthermore if the user is primarily interested in a detailed view of the camera-near points, the details are cached and rendered from near-to-far to show interesting details in front faster than in the background. Figure 3 shows the box-based partition steps.

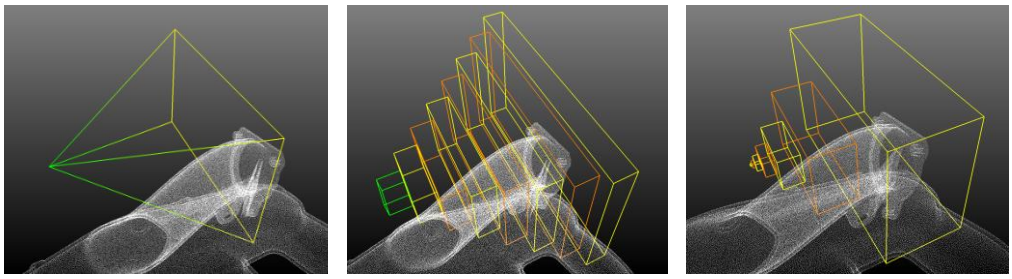


Figure 3. Illustration of a perspective frustum (left) and the frustum partitioning into axis-aligned boxes (center and right), which are used for the extraction of detailed data from the kd-tree. The center image shows a partitioning with constant box depth (green marks the camera nearest box) and the right image shows an exponentially increasing box depth

A smaller axis-aligned box can still include several millions of points. Therefore we use the overview point cloud to determine an adequate tree level for data querying with the following function:

$$Treelevel_{Details} = Treelevel_{Overview} + \log_2(\frac{N_{IP}}{N_{OPIF}}),\ with\ \ N_{IP} > N_{OPIF} > 0.$$

The tree level for querying the detailed point data depends on the ratio of the overview points already inside the frustum (N_{OPIF}) and the number of points to find (equals number of image pixels N_{IP}). When zooming into the point cloud, fewer overview points are inside the frustum and thus the tree level gets increased and more detailed data of the specific view volume is queued. If no overview point is inside the frustum due to a near zoom (but the bounding box of the tree is inside), the maximum tree level is chosen.

In contrast to the overview point cloud, we do not take all points from the computed tree level. Depending on the tree depth, there may be several hundred million points in a tree level. We use the axis-aligned boxes computed previously to query the detailed point data from the tree. The tree is recursively traversed from the root node and if the computed tree level is reached, the respective points are rendered and a deeper tree traversal is canceled. This way of tree traversal only addresses the subset of the potentially visible points in a tree level.

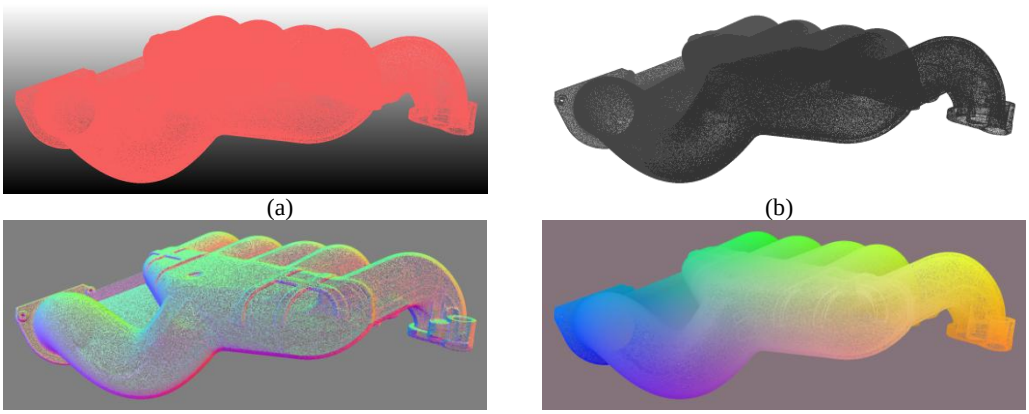
In addition to the classical frustum culling we apply a detail culling that uses the current detail volume. If all points of this volume, respective subtree, would be mapped onto the same pixel the kd-tree traversal is stopped.

3.1.3 Queuing Time and Blending LOD levels

Extraction of the detailed view may take some seconds depending on the entire data set size, but the operation is done asynchronously while the application stays responsive to user input. The time for detail extraction relies on various parameters like hard disk speed, access latency, data distribution and may vary heavily at different view positions. In order to apply feedback to the user, we display intermediary results, generated at discrete time intervals (e.g. every second). A smooth LOD transition requires little changes between frames to avoid popup effects and can be enabled by a geometric morphing approach (e.g. Grabner, 2001) or a blending approach like our proposed method in the section 3.2.2. We apply a fast image-based blending on the GPU that significantly reduces unwanted popup effects between different LOD resolutions. The detailed view is also cached on the GPU by using a Compute Shader (see section 3.2.3). This enables us to permanently show this view when moving the camera – only additional details are then blended into the existing view.

3.2 Deferred Rendering Methods

For rendering the point clouds we use a Deferred Rendering Pipeline (Pintus, 2011). Instead of rendering directly to the framebuffer we utilize a multi-pass approach consisting of a geometry pass followed by a lighting pass. In the first pass point attributes (color, normal vector, depth and world space position) are rendered into buffers (so called multiple render targets (MRT)). These are separate texture buffers for each point attribute (see figure 4). The lighting pass computes the shading for each visible pixel *deferred* in a shader program while rendering the scene into the framebuffer (we use a full screen quad).



(a) (b) (c) (d)

Figure 4. Multiple Render Targets of the Deferred Rendering Pipeline containing a color buffer (a), depth buffer (b), normal buffer (c) and a position buffer (d)

In our visualization approach we enhance this procedure by applying the following steps iteratively:

1. Extract a subset of points from the kd-tree that is located inside the viewing frustum (we use 100.000 points in a local point buffer for each iteration).
2. Render the points into the MRT, where the point attributes are stored in separate texture buffers. Within each iteration we only complement already existing information in the texture buffers (details are accumulated, see section 3.2.1).
3. Render a full screen quad to the framebuffer and apply the shading.
4. If all detailed point data is extracted then rendering is finished, otherwise continue again with step 1.

In order to achieve a good ratio of detail degree and processing time we extract points from a specific tree level only (corresponding to the number of screen pixels, see 3.1.2). Otherwise, we traverse the entire tree via breadth-first search if we prefer to generate a full detailed image, which has longer processing times.

This rendering method improves the LOD performance by at least 30 percent, because the CPU does not need to perform a time-consuming depth test and sends selected and probably visible points directly to the GPU which performs the Z-test extremely fast. This strategy reduces the work load of the CPU (for preselection of visible points) and utilizes the high bandwidth to the GPU and finally shortens the loading time of a detail level.

3.2.1 Detail Accumulation

If the camera is not moving, the render target will not be cleared and the cached detailed point data is rendered iteratively into the existing image (see accumulation scheme of the pipeline in Figure 5). The possibly visible points are cached stepwise from the kd-tree via breadth-first search and are then rendered.

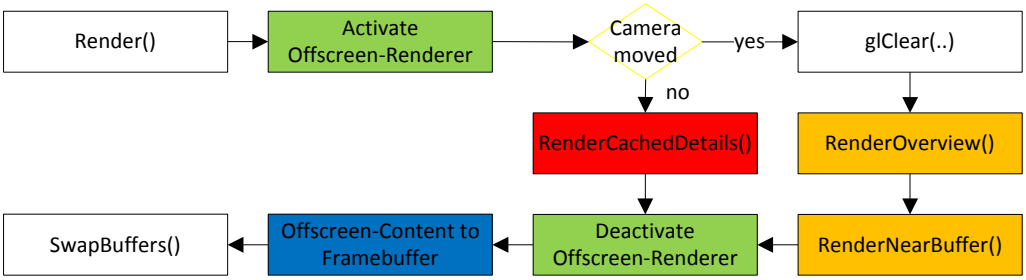


Figure 5. Scheme of the detail accumulation pipeline. All point data is rendered into an offscreen render target (green). If the camera has moved since last frame, the render target is cleared and the overview points as well as the cached detailed points from last frame are rendered (orange). If the camera stands still, only cached detailed points are added (red). Finally the content of the render target is rendered to the framebuffer for displaying the image on screen (blue)

This deferred detail accumulation pipeline allows a fully detailed view of the scene, even on low spec computer systems (OpenGL 2.X). When the camera position and orientation is constant between frames, no geometry is rendered twice, which results in high framerates. Figure 6 shows the detail accumulation from overview point cloud to full detailed point cloud.

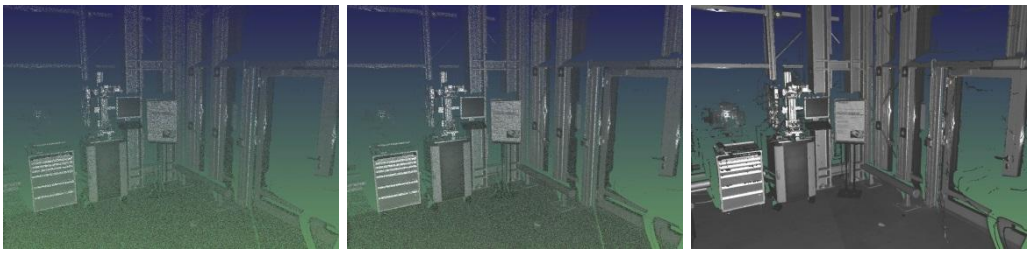


Figure 6. Detail accumulation for 650M points: Overview data is always rendered (left). If the camera is not moved, details are accumulated (middle and right image) whereat the transition is blended smoothly at constant time intervals

3.2.2 Image-based LOD-Blending

If the content of the render target is always shown immediately, the detailed data is continuously popping into the image. While this provides the user with instant feedback of the detail querying process, this can also be seen as very restless (and not preferred) visualization. In order to avoid the popup effects of the detailed data, we extended the rendering pipeline with two more render targets. Points from the kd-tree are rendered into one render target, which is never displayed on screen (a caching render target). The additional second and third render target are shown on screen implementing a double buffer strategy. At discrete time

intervals (e.g. one second) the images from the caching render target are copied to the second or third render target respectively. The visible change of the detail level is achieved via a continuous toggling and blending of the image from the second and third render target. This enables a computationally efficient and smooth LOD transition.

3.2.3 Detail Preservation

The data of the detailed view is always inside the images of the render target. To preserve this data and render it while the virtual camera is moving, the data in the render targets is copied into a Vertex Buffer Object. Figure 7 shows this operation enabled by a Compute Shader (available with OpenGL 4.3).

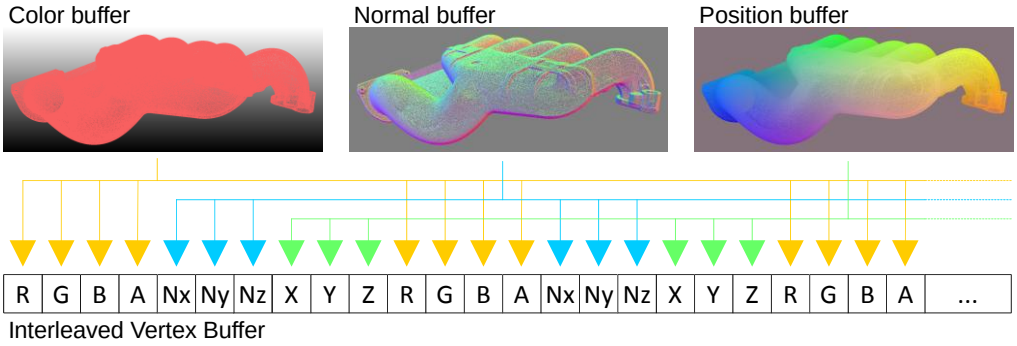


Figure 7. A Compute Shader copies the content of the color/normal/position buffer into an interleaved vertex buffer. Each pixel of the input images is written into the specific target positions of the vertex buffer

In our implementation, the detail preservation step is only done at discrete time intervals (as the LOD-Blending interval) and when all view-specific details are accumulated. By performing the caching and rendering of the last detailed view completely on the GPU, the detail preservation step is computationally efficient and fast.

3.3 Scheduling of Multiple Datasets

In our application we handle multiple point clouds with a scheduler mechanism. The scheduler provides a point stash for the overview point cloud, which is partitioned into N sections for N point clouds. The scheduler also provides a thread pool for the processing tasks of the data visualization.

3.3.1 Tasks

We split the operations for data processing and visualization into several tasks. A first task indexes the file and quickly generates an overview. The next tasks compute the leafs and the nodes of the kd-tree. Afterwards, a task re-computes the overview point cloud from the kd-tree. The overview re-computation is also done, when a new point cloud was added or if a point cloud was removed from the scene. A subsequent task is extracting the view-specific detailed point cloud data from the kd-tree. The processing functions (e.g. loading / storing a

kd-tree from file, smoothing, …) are also handled as separate tasks and enqueued into the task queue of the thread pool. Figure 8 shows the components of the scheduling concept.

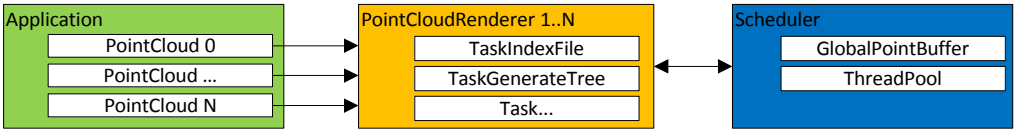


Figure 8. Each point cloud is handled by an instance of a PointCloudRenderer. Each PointCloudRenderer registers to a scheduler. The scheduler provides a global point buffer and a thread pool for processing all tasks in an ordered way depending on the resource usage (e.g. Hard disk I/O)

Each task has attributes assigned. There is a flag for addressing the hard disk directly (e.g. the file indexing task), a flag for using file mapping (implicit hard disk access) and an individual integer value for the priority. The file indexing task has the highest priority value to provide an overview of all data sets as fast as possible. All other tasks are considered with lower priority. The detail extraction task has an integer range where the distance of the virtual camera to the point cloud is mapped onto the priority value to extract data of nearby objects earlier.

3.3.1 Prioritization and Ordered Processing

The tasks are processed in an ordered way by considering their type and attributes. The thread pool is initialized with a number of threads equal to the number of CPU-Cores minus one for the main application. Each thread in the thread pool repeatedly accesses the task queue to grab a task for processing. A thread scans at first for the task with the highest priority value. If the task attribute indicates a direct access to the hard disk and another thread already uses the hard disk, the task will not be chosen and the next will be considered. If the task uses file mapping, the free physical memory is considered further. If all tasks left are detail extraction tasks and there is free physical memory avail, the task will be chosen because the tree is completely mapped into the memory. Otherwise the task will be chosen only if no other thread is processing a task with file mapping. The concurrent (and in total slower) multiple access of the hard disk is suppressed by this approach.

The thread pool processes the time consuming tasks asynchronously and implicit to the main application render thread. Besides from taking advantage of multi-core CPUs, this also ensures that the main application thread itself has always short response times to user input. The progress of the operations in the background is displayed to the user with a progress bar to provide continuous feedback. The concept of ordered processing enables a smooth exploration of large data sets independent from its size.

4. RESULTS AND DISCUSSION

We tested our implementation on a standard pc system, which contains an Intel i5-4670 processor, 8 GB RAM, a Nvidia GeForce GTX 760 graphics card and an Intel SSD with 180 GB. All test point clouds are stored in an Ascii-Format, where each line contains 3 floats for the position and three integers for the color of a point. We used three data sets for the evaluation, a manufactured object with 65 million points (as in figure 4), an indoor laser scan

with 233 million points (figure 6) and a 3d point cloud from earth data with 1 billion points (figure 1). This represents the fields of application for our technology, the 3D-Inspection and Virtual Walkthrough of large 3d point clouds.

The rendered overview point cloud in each test case has 8 million points. When moving the camera, we always achieve a high interactive framerate with at least 150 fps at a 1920 x 1080 pixel resolution. We achieve up to 4000 fps, when the camera stands still, since detailed points are only added and are not rendered multiple times. This results in a very low input lag as one would expect from computer games. Nevertheless, the process of successively extracting all the points which are currently visible typically takes less than a second and depends on the file I/O performance of the system. In order to steadily provide feedback to the user we show intermediate results at discrete time intervals (with a smooth transition via image-based blending). Table 1 shows the processing time for individual steps of multiple test cases.

Table 1. Processing times for example point clouds. Data set 1, 2 and 3 are single point clouds. Data set 4 is the data set 1 divided into 8 parts with nearly same point size (to evaluate the scheduling mechanism for multiple point clouds). Data set 5 is the divided data set 2. The tree building time is much lower in the divided data sets, due to the scheduling system. The time for the visible full detailed view depends on the view position and ranges from milliseconds to seconds. In typical viewing scenarios the full detailed view is available in less than a second. The last row of the table shows the maximum time measured from multiple different viewpoints

	Data set 1	Data set 2	Data set 3	Data set 4	Data set 5
Points	65m	233m	1bn	8 x 8m	8 x 29m
File size	2.6 GB	9.9 GB	42.2 GB	8 x 0.3 GB	8 x 1.2 GB
Index xyzc-file (total)	7.96 sec	29.91 sec	126.72 sec	7.90 sec	32.07 sec
Build kd-tree (total)	23.69 sec	260.90 sec	1691.31 sec	18.77 sec	78.19 sec
Detailed view (max)	6.02 sec	15.11 sec	46.71 sec	4.11 sec	14.95 sec

While the processing of the data sets is done asynchronously and the interactive exploration is not limited by the data size, we decided to extend the test to a less powerful computer system. We evaluated our approach on a laptop with an Intel i5-5200U (2 processor cores), 8 GB RAM, an integrated Intel HD 5500 graphic card and an external HDD with 1 TB. We tested all data sets and always achieved at least 35 fps at a 1366 x 768 resolution. The asynchronous data processing concept enables the usage of our method in scenarios where the exploration of scan data sets needs to be done while the 3D scanner is capturing data in parallel.

Additionally, the detail accumulation technique retains more processing power of the GPU by not rendering geometry multiple times. Thus, we are able to apply more complex lighting models like physically based shading to enhance the visualization. Our accumulation technique is also not limited to render static scenes. Via Shading Language it is possible to mix the images with a dynamic scene (e.g. live sensor data).

5. SUMMARY

In this paper we have presented techniques for a smooth visualization of large 3D point clouds with up to a billion of 3D points utilizing an out of core data storage. In addition to improved technologies for structuring the raw data by using trees and memory mapping techniques for accessing large data sets, we provided enhancements to existing LOD techniques. We extended the Deferred Rendering Pipeline to accumulate details and utilized Compute Shaders to preserve the details while moving the virtual camera. Furthermore we applied a smooth transition of LOD levels / detail queuing by extending the render pipeline with image-based blending. Conclusively we introduced a scheduling system to handle and render multiple point clouds with interactive framerates. The techniques proposed extend existing approaches for the visualization of large volume data and large polygonal data to unstructured raw point data from optical 3D scanners. At the example of real test data sets we have shown that the implementation enables a fast and smooth exploration of large point clouds on a standard computer system with interactive framerates.

REFERENCES

Boehm, J., 2014. File-centric Organization of large LiDAR Point Clouds in a Big Data context. In *Workshop on processing large geospatial data*, Cardiff, UK

Botsch, M. et al., 2005. High-Quality Surface Splatting on Todays GPUs. In *Proceedings of the Second Eurographics / IEEE VGTC Conference on Point-Based Graphics*, pp. 17-24

Dementiev, R. et al., 2005. STXXL: Standard Template Library for XXL Data Sets. In *Algorithms - ESA 2005*, Springer Berlin Heidelberg, pp. 640-651

Ferko, M., 2012. *Real-time Lighting Effects Using Deferred Shading: Creating a System for Real-time Rendering Without Relying on Pre-computation*. Lap Lambert Academic Publishing

Futterlieb, J. et al., 2016. Interactive Visualization of Massive 3D Point Clouds. In *Proceedings of the International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing*, pp. 223-230

Garg, R. P., Sharapov, I., 2001. *Techniques for Optimizing Applications: High Performance Computing*, Prentice Hall PTR, pp. 425-431

Goswami, P. et al., 2012. An efficient multiresolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees. In *The Visual Computer Volume 29, Issue 1*, pp. 69-83

Grabner, M., 2001. Smooth High-quality Interactive Visualization. In *Spring Conference on Computer Graphics*, pp. 87-94

Pintus, R. et al., 2011. Real-time Rendering of Massive Unstructured Raw Point Clouds Using Screen-space Operators. In *Proc. of the 12th International Conference on Virtual Reality, Archaeology and Cultural Heritage*, pp. 105-112

Ribelles, J., et al., 2010. An Improved Discrete Level of Detail Model Through an Incremental Representation. In *EG UK Theory and Practice of Computer Graphics*, pp. 59-66

Richter, R. et al., 2015. Out-of-core Visualization of Classified 3D Point Clouds. In *3D Geoinformation Science - The Selected Papers of the 3D GeoInfo 2014*, Springer International Publishing, pp. 227-242

Rusinkiewicz, S., Levoy, M., 2000. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 343-352

Sunar, M. S. et al., 2006. Accelerating Virtual Walkthrough with Visual Culling Techniques. In *International Conference on Computing & Informatics (ICOCI 2006)*, pp. 396-400

Teutsch, C., 2007. *Model-based Analysis and Evaluation of Point Sets from Optical 3D Laser Scanners*. Shaker Verlag GmbH, Germany

Weller R., et al., 2013. Parallel collision detection in constant time. In *Virtual Reality Interactions and Physical Simulations (VRIPhys)*, Lille, France, Eurographics Association, pp. 61-70

Wenzel, K. et al., 2014. Filtering of Point Clouds from Photogrammetric Surface Reconstruction. In *ISPRS – International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume XL-5*, pp. 615-620

Wimmer, M., Scheiblauer, C., 2006. Instant Points: Fast Rendering of Unprocessed Point Clouds. In *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*, pp. 129-136