Eurographics Symposium on Point-Based Graphics (2007)
M. Botsch, R. Pajarola (Editors)

Interactive Editing of Large Point Clouds

M. Wand[1], A. Berner[2], M. Bokeloh[2], A. Fleck[2], M. Hoffmann[2], P. Jenke[2], B. Maier[2], D. Staneker[2], A. Schilling[2]

[1]Max-Planck Center VCC, Stanford University, USA
[2]WSI/GRIS, University of Tuebingen, Germany

Abstract

This paper describes a new out-of-core multi-resolution data structure for real-time visualization and interactive editing of large point clouds. In addition, an editing system is discussed that makes use of the novel data structure to provide interactive editing tools for large scanner data sets. The new data structure provides efficient rendering and allows for handling very large data sets using out-of-core storage. Unlike related previous approaches, it also provides dynamic operations for online insertion, deletion and modification of points with time mostly independent of scene complexity. This permits local editing of huge models in real time while maintaining a full multi-resolution representation for visualization. The data structure is used to implement a prototypical editing system for large point clouds. It provides real-time local editing tools for huge data sets as well as a two-resolution scripting mode for planning large, non-local changes which are subsequently performed in an externally efficient offline computation. We evaluate our implementation on several synthetic and real-world examples of sizes up to 63*GB.*

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - Object Hierarchies; I.3.5 [Computer Graphics]: Graphics Utilities - Graphic Editors

1. Introduction

Acquisition of real-world scenes using 3D scanning technology is nowadays a standard technique. Applications range from acquiring small mechanical parts for quality control to acquisition of archaeological sites or even complete cities [AFM*06]. In such applications, enormous quantities of 3D data are produced, typically in the form of large unstructured point clouds. Usually, it is necessary to perform further processing on the acquired data. Typical editing operations are either manual interactive editing (for example, removing artifacts caused by people walking by from a city scan), or automatic processing such as filtering and normal estimation. Given the capabilities of modern commodity hardware, processing medium sized scenes (up to some hundred MB in size) is not problematic and several tools have been developed that specifically target handling point cloud data directly [ZPKG02, PKKG03, WPK*04]. However, editing and geometry processing becomes much more problematic if data sets exceed the limits of main memory. There exist a well developed set of techniques for offline streaming processing of large quantities of geometry [IG03, CMRS03, IL05, Paj05]. However, currently no techniques exist that allow interactive editing of large point clouds. The main challenge here is the coupling of visualization and processing: Traditional techniques for real-time display of large scenes require expensive preprocessing to build the data structures that accelerate rendering. Rebuilding such data structures after modifications of the scene is too costly for interactive applications. In consequence, it is not possible to perform interactive editing of scenes that are substantially larger than available main memory. In this paper, we propose a new data structure that fills this gap: We describe a point-based multi-resolution data structure that allows for fast rendering of complex scenes, efficient online dynamic updates and out-of-core storage of unused portions of the data set. The complexity of a dynamic update operation is proportional to the amount of geometry affected while the size of the scene has only minor influence on the update costs. In addition, the dynamic update algorithms have been carefully designed for high throughput so that local editing can be performed in real time: A user walking through the virtual scene is able to change portions of geometry of up to some hundred thousand points interactively and immediately continue his inspection of the data set. For even larger editing operations, the performance degrades gracefully (i.e. linearly) to streaming out-of-core processing. The only general requirement is spatial coherence, i.e., the actual data blocks accessed within a local time window must fit into main memory. During processing, a full multi-resolution representation

of the complete data set is immediately available. We have implemented the data structure as part of an interactive editing system for large point clouds. The architecture of the system has been specifically optimized to complement the features of the underlying data structure. It uses a command history architecture [Mye98] that allows for planning complex, global editing operations on a simplified low resolution model and then applying the operations to the final high resolution data set in an offline process. We evaluate the performance of the new data structure on both synthetic and real-world examples. In practice, we have been able to perform real-time walkthroughs and interactive local editing in scenes of up to 63GB in size. The implementation of the data structure and the editor application are available online as open source software [XGR07].

2. Related Work

In literature, many solutions have been proposed for efficient rendering and processing of complex geometry. An exhaustive summary is beyond the scope of this paper. In the following, we will therefore mostly focus on out-of-core techniques that allow for handling large quantities of data as well as large scene editing techniques. The basis of most rendering strategies for complex geometry is geometric simplification. For triangle meshes, this is typically done using edge contraction algorithms [Hop96]. To avoid dealing with triangulation consistency issues, early out-of-core simplification algorithms [Lin00, SG01, Lin03, SG05] have been based on vertex-clustering [RB93]. Our data structure employs a similar grid-clustering technique to provide efficient multi-resolution updates. General out-of-core edge contraction can be implemented using blockwise simplification and edge stitching [Hop98]. In order to efficiently utilize current graphics hardware, which works batch oriented [WH03], fine grained hierarchies are not optimal. More recent approaches therefore form coarser hierarchies by combining spatial subdivision and simplification within each spatial subdivision cell [CGG*04, GBBK04, YSGM04, BGB*05]. We adopt a similar idea to hide latencies in our approach. Recently, point-based multi-resolution data structures have become popular because of their robustness in cases of strong topological simplification and not at least because point-based representations are the native format of most 3D acquisition devices. Point-based multi-resolution representations have been proposed with the Surfels and the QSplat systems [PZvBG00, RL00]. The first forms an octree hierarchy of sample points with fixed relative spacing, while the second is based on a bounding sphere hierarchy which itself provides the splats. Streaming QSplat is a network implementation that provides external efficiency by utilizing a layered memory layout and blocking [RL01]. Other out-of-core algorithms have been built on top of the Surfels idea of using large point clouds per node in a spatial hierarchy to achieve external efficiency [GM04, WS06]. The data structure proposed in this paper follows this line of work. None of the aforementioned approaches supports efficient dynamic changes; the data structures are built once in an offline preprocessing step. Up to now, only very few schemes have been proposed that allow dynamic modifications. [WFP*01] describe a point-based multi-resolution rendering scheme that supports dynamic updates. However, the employed randomized sampling pattern cannot be implemented efficiently in out-of-core settings. Klein et al. [KKF*02] develop a similar sampling-based representation for out-of-core settings but do not provide dynamic operations. In recent work, [BS06] describe a two-resolution scheme for texturing and painting on large out-of-core models. In contrast to our method, this technique is restricted to attribute modifications, not allowing the geometry to be changed. A multi-resolution editing system for large out-of-core terrains based on wavelets is described by Atlan and Garland [AG06], similar in spirit to earlier (in-core) wavelet-based image editing systems [BBS94]. The wavelet approach is clean and elegant but the heightfield assumption does not allow for handling objects of arbitrary topology. Multi-resolution editing for surfaces of general topology has been described earlier by [ZSS97], however, without out-of-core capabilities so that only moderately complex surfaces can be handled. A similar multi-scale editing approach for point clouds has been proposed by Pauly et al. [PKG06], again not targeting at large scenes. Another large scene editing approach has been described by Guthe et al. [GBK04], who dynamically create a triangle hierarchy from a NURBS scene description. An alternative, non-hierarchical approach for processing huge data sets is streaming processing [IG03, IL05, Paj05]. Here, the processing order is fixed and data is processed in one or more sequential passes. This processing model allows highly efficient algorithms but does not allow general data access, therefore not permitting interactive visualization. A data structure for more general out-of-core processing of triangle meshes using spatial hierarchies has been described by Cignoni et al. [CMRS03]. However, this data structure does not provide a multi-resolution representation, which is necessary for interactive rendering. Our system was strongly motivated by the PointShop3D editing system of Zwicker et al. [ZPKG02] for point cloud processing. PointShop3D aims at moderately sized, in-core data sets. The goal of our work is not to provide a similarly rich set of point cloud editing tools but rather demonstrate that point-based editing techniques in principle can be applied to large data sets. Most point-based geometry processing techniques [PKKG03, WPK*04] rely heavily on local proximity queries such as k-nearest neighbors or geometric range queries. We address the problem of providing efficient support for fine grained geometric queries as well as efficient rendering using a layered approach of nested hierarchies that are updated transparently during editing.

3. The Dynamic Multi-Resolution Out-of-Core Data Structure

In the following, we will discuss our new data structure. It consists of a dynamic octree with a grid-quantization-based dynamic multi-resolution representation in each inner node. Subsequently, we describe how the data structure and the dynamic operations are implemented and how out-of-core data storage is realized. After that, we will briefly address the concept of secondary data structures and the employed rendering strategy.

3.1. Multi-Resolution Data Structure

Our basic data structure is an octree [PZvBG00], which is especially well suited for dynamic operations due to its regular structure. All octree nodes are associated with cubes that are bounding volumes of the geometry stored in their corresponding subtrees; the root node is associated with a cube that contains the whole scene. We store all data points in the leaf nodes only and inner nodes provide simplified multi-resolution representations. The subdivision depth of the tree is uniquely defined by requesting that no leaf node should contain more than n_{max} points (typically: $n_{max} \approx 100,000$) and no unnecessary splits should be performed, i.e. no set of leaf nodes should exist that can be subsumed into the parent node without exceeding n_{max} points per leaf. Updating such an octree dynamically is a standard problem [Sam90]. The main contribution of this paper is a dynamic multi-resolution scheme for such spatial hierarchies that can be updated efficiently. The basic idea is to provide downsampled point clouds in inner nodes with a sample spacing that is a fixed fraction of the bounding cube side length of the corresponding node ([PZvBG00], see Figure 1). We create these downsampled representations using a quantization grid [RB93, WFP*01]: In each inner node of the octree, we establish a k^3 grid (typically $k \approx 128$). All points of the subtree are quantized to the grid cells and only one representative point is kept. In order to keep track of the representation under dynamic modifications, we also store a weight for each grid cell that counts the number of points represented (Figure 2a). This weight is necessary for dynamic modifications later on. For handling these weights, we have different choices: The first option (which is used in our current implementation) is to just store a random representative point and count the number of points stored in the weight variables. To improve the uniformity of the sampling pattern, we can also quantize the points stored in the grid cells. To avoid aliasing, we could also store a sum of all point attributes instead of just a single representative point and then normalize the average by dividing by the weight counter. This corresponds to an unweighted average filter. The quantization grid itself is not stored explicitly as an array but using a hash function: The quantized x, y, z-positions of the points are input into a pseudo-random hash function to access buckets in a dynamic hash table. The length of the bucket list is adjusted dynamically to retain a constant fill factor (1 in our current implementation). Using this representation, only O(m) bytes of memory are used for storing m non-empty cells and random access is in expected constant time. In out-of-core settings, the hash tables are not stored in disk but rebuild on-the-fly from a list of non-empty cells.

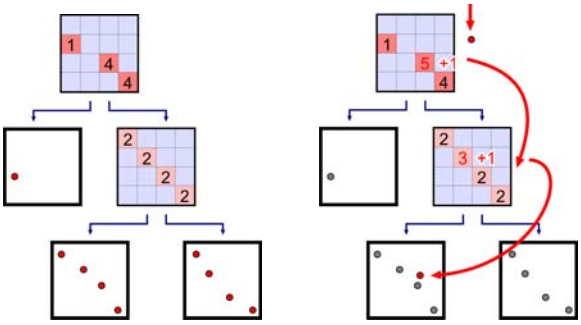
Figure 1: *The first three levels of a surfel hierarchy.*

3.2. Dynamic Updates

Grid quantization is not the best uniform sampling strategy [Wan04] and also does not provide adaptive sampling [PGK02], which potentially reduces the sampling cost further. The big advantage, however, is that this technique permits efficient dynamic updates. Our data structure provides two basic dynamic operations: insertion and removal of points from the multi-resolution tree. More complex operations are mapped to a sequence of removals and (re-) insertions. In the following, we describe the dynamic operations on the octree and how the multi-resolution data structure is kept up-to-date during these operations.

Inserting points: If the data structure is empty, a leaf node containing the single point is created (leaf nodes do not store a multi-resolution grid, only inner nodes do). Otherwise, two cases have to be distinguished: the new point might be inside or outside the root node of the tree. If the point falls inside the root node, we determine a path to the lowermost node in the tree that contains the point. If this is a leaf, we insert the point into the leaf. If the last node containing the point is an inner node, we create a new leaf for this point. On the way down from the root node, we insert the point into each quantization grid in each inner node we visit (Figure 2b). To do so, we only have to compute the quantized grid cell, add/store the attributes of the points and increment a counter. Therefore, the overhead in comparison to a standard dynamic octree that does not keep a multi-resolution representation in its inner nodes is small. Please note that the update costs do not depend at all on the complexity of the multi-resolution representation in each node, every update is O(1) per node. After the update, we run an amortized validation step that subdivides nodes if necessary; this step is discussed later as it will be used in subsequent operations as well. In order to guide the validation procedure, we also mark the path down from the root node with flags. In the second case, where the point falls outside the root bounding box, a new root box has to be created. In order to do this, we enlarge the root box by a factor of two and make the old root a child of the new one. We chose one of the 8 possible new positions for the root box that brings the new box closest to the new point. This procedure is repeated until the new point

is comprised in the root box. Then, a new leaf below the root box is created to store the new point. Now we have to update the multi-resolution representation in the new root node and the chain of nodes leading to the old root node. We use the same grid-quantization algorithm as in the previous case on these nodes to update their quantized point sets, inserting the multi-resolution representation stored in the quantization grid of the old root node and the new point as well into the nodes on top of the old root. For consistent counting, we add the correspondingly larger weights when processing points from the quantization grid of the old root.



(a) quantization tree with counting

(b) inserting a point: incrementing counters

Figure 2: *Updating the multi-resolution representation*

Deleting points: Deletion of points proceeds in a similar way. We delete the point from the list in its leaf node and then follow a path upward the hierarchy to the root node. In each inner node, we compute the quantized grid cell this point corresponds to, decrement the weight by one and, optionally, subtract the point attributes if running averages are used. If the weight becomes zero, the entry is removed from the hash table storing the quantization grid. Again, the nodes on the path are marked for validation. To locate a point, i.e. if only its position but not its node and list index are known, we trace a path from the root node to the leaf node that contains the point and linearly search the leaf list for the point position.

Validation step: The algorithms outlined above are not yet complete because they do not enforce the n_{max} criterion for the size of the leaf nodes. To do this, we run an additional validation step. This validation could be performed after each point insertion but in practice this severely degrades performance (typically by an order of magnitude). Therefore, we call the validation algorithm only after a large number of points have been changed or before a new rendering operation is invoked. "Large" means that we must prevent overflows of the leaf node lists in out-of-core settings, so we typically perform validations after a maximum of 100,000 points have been inserted. The validation algorithm follows (and clears) the flags down to modified leaf nodes and then performs two checks: If a leaf node contains more than n_{max} points, it needs to be split. This is done by considering this leaf node a root node of a local sub-octree and performing the dynamic insertion algorithm as outlined above, including the updates of newly created multi-resolution grids. At this point, the former leaf node will become an inner node and will be assigned a multi-resolution grid as well. The second check refers to the opposite case: Whenever a parent node of a leaf node contains a subtree that contains no more than n_{max} points, the whole subtree can be compressed into one new leaf node. All the other nodes and the multi-resolution representation of the parent node are discarded in this process. After such an operation, the same check is applied recursively to the newly created leaf node and its parent until no more node removals are possible. A subtlety here is to efficiently determine the size of a subtree of an inner node. For this, we keep track of the number of points in each subtree during all dynamic operations by counting: Each node contains a 64Bit counter that is updated when points are traced through the hierarchy during insertion and deletions. Counting is not limited to counting the number of points; our implementation also supports counting other subsets of points, for example those flagged for selection. This mechanism will be used later on for efficiently retrieving selected points in large point clouds by recursively traversing only subtrees that contain selected points.

Discussion: We have also considered an alternative design of the data structure, where points would only be inserted and deleted into/from the leaf nodes and the multi-resolution representation would be rebuilt lazily, bottom-up on demand. This approach would have had the advantage that adaptive sampling and more general attribute filters in multi-resolution nodes could be implemented more easily. Aiming at real-time editing, we opted for the scheme presented here because interactive edits (typically affecting only small portions of the points of a leaf node) are expected to be much more efficient with the chosen strategy.

3.3. Out-of-Core Storage

The multi-resolution representation described above can easily be used in out-of-core settings. In the following, we basically apply standard virtualization techniques [RL01, KKF*02, CGG*04, GBBK04, YSGM04, GM04, GM05, BGB*05, WS06] to the data structure described above: The main idea is to load only those nodes into memory that are needed for rendering or editing. For changing a single point of the original data, only a path from the root node to a leaf node has to be accessed, which, in practice, is a very small number of nodes. For rendering with fixed on-screen resolution, only a small fraction of nodes has to be accessed while most of the data structure remains unused. Our implementation provides two methods for out-of-core support: fetch and access. Fetching demands a node to be used in the near future and access asserts that the data resides in main memory. We use a standard LRU (least recently used) queue to swap out unused nodes to disk when the allocated local cache for nodes is exceeded. Nodes that have not been mod-

ified are deleted immediately and not written back to disk. Using 64Bit integers for node indices, we can handle scene sizes that are practically only limited by available secondary memory. Fetching of nodes is done in a separate thread to hide disk latencies. Fetch operations and in-core processing can be performed concurrently, only actual access to non-memory resident nodes is blocking. The nodes themselves are stored in multiple files, each file providing a stream of blocks of a fixed size, with block-sizes increasing in powers of two among the files. Using this approach, every block can be directly accessed with only one seek operation. Using multiple files leaves the allocation and fragmentation problem to the operating system, which worked reasonably well in our experiments. In case more explicit control over disk space allocation is needed, one could also allocate big blocks and subdivide them in powers of two to create allocation units that are guaranteed not to be fragmented, however, at increased allocation and implementation costs.

An interesting question that remains is whether the hierarchical out-of-core data structure is efficient, and how we should choose the parameters for best rendering and processing performance. The important trade-off here is between blocking overhead and block access latencies: A typical off-the-shelf hard disk has a random access seek time of roughly 10ms and a transfer rate of about 50-100MB/sec. This means that reading 500KB blocks from disk wastes 50% of the available bandwidth due to latencies. Thus, node sizes should be chosen in the range of hundreds of kilobytes or megabytes at least. Obviously, using larger block sizes increases the throughput. On the other hand, using large blocks has disadvantages due to the reduced adaptivity. For editing, it is harder to track the region of interest precisely with coarse blocks. Similarly, for rendering, view frustum culling and level-of-detail control become less accurate for larger block sizes. In the case of localized editing, the minimum requirement is to store at least 1-8 adjacent leaf nodes (in case boundaries regions are involved) and a path to the root node in order to perform local operations efficiently, which is easily met. For larger editing operations, the actual performance depends on the data access patterns. For rendering, the trade-off of using point clouds of different sizes in a multi-resolution octree is analyzed in [Wan04]. It is shown analytically and empirically that even larger nodes (typically 20% or more of the screen size under projection) can be used with acceptable overhead. For VGA resolution, this corresponds to quantization grids of roughly $k \approx 128$, which is sufficient to hide latencies (for our replicated bunny benchmark scene, see Section 5, this corresponds to an average of 750KB per inner node for position only and estimated 1.3MB with color and normals).

3.4. Secondary Data Structures

For out-of-core paging, we have to use a very coarse hierarchy. This is not favorable for operations such as small scale range queries or nearest-neighbor queries, which are frequently used in point-based modeling. Therefore, we create a secondary layer data structure that can be attached to nodes in the primary hierarchy. In our current implementation, we use octrees with a node size of $n_{max} = 20$ points that are attached to the leaf nodes of the hierarchy. For the query algorithm, we treat the cascaded hierarchy as one single virtual hierarchy. In general, secondary data can be stored on disk along with the main hierarchy. For the secondary octrees, we currently do not use this option but delete them when being paged out and recompute them on demand. The current implementation uses an efficient iterative static octree construction algorithm that processes about 1 million points per second, which is almost as fast as loading from disk and does not use additional external memory. The secondary hierarchy stores only indices to the points in the primary hierarchy's node and is recomputed on demand after changes to the geometry. However, conceptually, other data structures such as secondary kD-trees or dynamic octrees could also be handled easily using the same mechanism. The secondary data structure mechanism is also useful for rendering: Depending on the performance characteristics of the external memory versus those of the graphics card, different blocking factors might be favorable for the two domains. If rendering works better with smaller block sizes, we can use the secondary octree to achieve more adaptivity while still fetching larger blocks from hard disk. However, for current hardware it turned out that this is not necessary: Current GPUs have a per-batch latency of about 10μs (Windows XP, [WH03]) and internal bandwidth in the range of up to 50-100GB/sec. This is three orders of magnitude faster than the external storage (a single, contemporary consumer hard disk) but shows almost the same ratio between latency and throughput, which determines the optimum block size. In addition, given the performance gap, the hard disk will typically be the bottleneck, even if slow movement of the observer allows for extensive caching. Therefore, we currently use the primary hierarchy for both rendering and paging and optimize the parameters for optimum hard disk utilization, which typically also gives good GPU throughput. In case a drastically different hardware setup is employed (such as RAID arrays with higher throughput/latency ratio and/or new GPU APIs with drastically reduced latency), secondary octrees give us the necessary flexibility to adapt the block sizes.

3.5. Rendering

The rendering algorithm performs a depth-first traversal of the hierarchy until the minimum projected point spacing in an octree box is below a fixed threshold [PZvBG00] (typically 1-5 pixel). During the descent, the algorithm tests nodes for main memory presence and schedules non-present nodes to be fetched by the background thread, using a lower resolution representation for rendering until the nodes become available. For local image reconstruction, we use point primitives (point sprites) that are scaled according to the point sample spacing [RL01]. The minimum primitive size

is assumed to be given in the original data and enlarged automatically for inner nodes of the hierarchy in order to prevent reconstruction holes. In case this information is not available, we compute it from k-nearest neighbor distances as described below.

4. System Architecture

We have implemented an interactive point cloud editing application on top of the proposed data structure that provides prototypical editing tools (selection, transformation, painting, etc.) as well as batch processing operations such as PCA-based normal and sample spacing estimation. In the following, we give a brief overview of the system architecture and how scalable, output-sensitive editing tools can be implemented.

4.1. Channels, Iterators and Tools

The basic primitive of the system is a point cloud, which is defined as an unstructured set of points where each point provides a set of attribute channels such as position, color, or normals, which can be configured at run time. Within one point cloud, all points have the same set of attributes. For rendering, the channel values are copied into GL vertex buffer objects and are available to the vertex shader of the GPU. Access to data is not granted directly but via iterators that abstract from a possible out-of-core implementation (the system provides non-hierarchical and hierarchical point clouds, the latter in an out-of-core and an in-core version). Iterators can be hierarchical or linear; the first represents a hierarchy of boxes containing lists of points while the second provides only linear access to a virtual list of points. Following the paradigm of functional programming, iterators can be cascaded. For example, the basic iterators expose the primary hierarchy. They are used as input to a cascaded iterator which automatically detects, creates and attaches secondary octrees on-the-fly in leaf nodes and transparently returns a deeper, secondary hierarchy. A third iterator might use the previous cascade to perform k-nearest neighbor queries, resulting in a virtual long list of nearest neighbors to a geometric position, which then has a linear, non-hierarchical structure.

4.2. Implementing Editing and Processing Tools

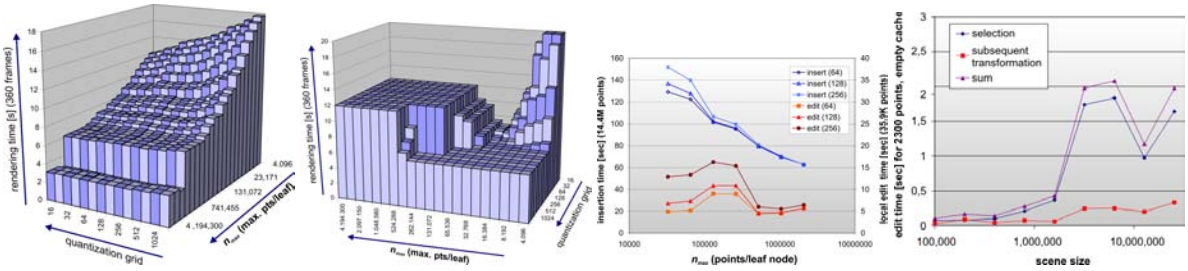
Our current editor application uses cascaded iterators for general geometric range queries (filtering out points and subtrees not intersecting a given subsets of three space), for k-nearest neighbors computation (as described above) and for selection (filtering out non-selected points hierarchically). Using these building blocks, we have created interactive tools for selecting, painting, texturing, transforming and deleting points. The point selection tool is for example implemented by issuing ray queries, i.e., range queries for perspective extrusions of on-screen circles or rectangles. For points within this range, a selection flag is set. Surface painting, texturing and embossing are implemented by first determining the closest point within a narrow ray and then issuing a spherical range query around the point of intersection. Actual modifications to data are implemented by deleting and reinserting points with modified attributes. In case the position has not changed, a more efficient implementation is used that changes only the point attributes and updates the multi-resolution hierarchy accordingly, avoiding reinserting points, which speeds up tools such as surface painting. For deleting selected points, we finally use the hierarchical selection iterator to efficiently retrieve selected points and remove them from the data structure. All operations described so far are strongly output-sensitive; the running time for local editing is $O(mh)$ where m is the amount of data affected (up to node accuracy) and h is the height of the hierarchy. The scene size affects the execution speed only via the hierarchy height h, which, in practice, can be expected to be logarithmic in the number of primitives and thus reasonably small.

4.3. Command Objects

The tools described before are not efficient for large scale changes. For example, painting on 20 million points at a time cannot be done at interactive rates. It seems that there is no principal solution to this problem, as processing a large number of points will inevitably take a long time. However, we can improve the usability in these cases by adapting the software architecture. In our editor application, we have implemented a command object architecture [Mye98] that automatically records all user actions and stores these as a list of commands. Each command consists of a set of parameters, including user input such as the parameters of the query region for selecting points or the color of painting operations. During interactive editing, the user can browse the list of commands, change parameters and reexecute subsets of the commands. We have also implemented a simple scripting language that provides variables and basic control flow and is able to alter parameters and reexecute previously recorded commands procedurally. These capabilities can be used to perform large scale edits: At first, editing is performed on a low resolution version of the scene for which global editing is efficient. For this purpose, we have implemented a simple streaming-simplification tool that selects a fraction of points randomly. The user can then perform operations interactively and save the resulting command script. Then, by removing the resampling command and reexecuting the script, the full resolution edit is computed offline. All previously described editing tools are externally efficient; if the selection comprises a large set of points, these will automatically be processed in octree-order which leads to coherent processing patterns and avoids cache trashing.

5. Results

In order to evaluate our proposal, we have conducted a series of experiments on synthetic and real-world data sets. Unless noted otherwise, all tests have been performed on a mid-



(a) hierarchy creation / editing, in-core (b) rendering, in-core (c) hierarchy creation / editing, out-of-core (d) editing, increasing scene size

Figure 3: *Processing times for different parameter settings. All parameter axis are shown in log scale, timings in linear scale.*

range PC system with single core Pentium-4 3.4GHz with Hyperthreading, Seagate 500GB/7200rpm SATA hard disk and an nVidia Quadro FX 3450 graphics board.

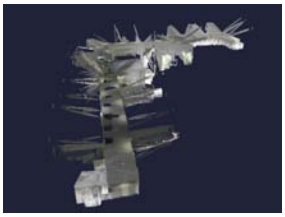
Parameter optimization: In order to determine suitable parameters for the data structure, we create a simple synthetic test scene for various parameter settings and measure the performance of rendering and editing operations. In the first test, we use a script that replicates a number of Stanford bunny models on a simple heightfield (2,2M points total) and renders a walkthrough. The results are shown in Figure 3: As expected, hierarchy creation becomes more expensive with increasing depth, i.e. smaller leaf nodes. Additionally, the quantization grid also has an influence; for fixed n_{max}, larger grids are more expensive to build, as this increases the data structure size (in practice, we choose parameters so that leaf and inner node sizes are comparable). At typical in-core parameters ($n_{max} = 131,072$, $k = 128$), we can insert 300,000 points/s. For rendering, larger batches, i.e. large quantization grids and many points per leaf node, are better. Only for very large leaf nodes ($> 500,000$ points), performance degrades. The effect is limited in this example due to the small scene size. Next, we perform a similar test using the out-of-core hierarchy. Figure 3c shows hierarchy creation (14M points) and local editing times (working on 36K points) for the same test scene. Again, construction times drop for coarse grained hierarchies, where local editing times are optimal for about $n_{max} = 500,000$ points.

Output-sensitive editing: Next, we perform a fixed-size editing operation (selecting and subsequently moving a patch of ca. 2300 points out of a large flat quad covered uniformly with points) for increasing scene size (Figure 3d). Scene creation and editing are performed subsequently without flushing the memory cache. Correspondingly, the editing time goes up significantly once the whole scene does not fit into the main memory cache anymore. The expensive part is the first selection command, which pages the nodes into main memory. The subsequent transformation command, which accesses the same points again, is much more efficient. Overall, editing time for cached geometry (which is the typical case for local edits) changes only moderately with scene size; local editing can still be performed efficiently in very large scenes (the expensive first editing operation is an artifact of the synthetic, non-interactive benchmark; typically, one would only select geometry that has been previously fetched for rendering).

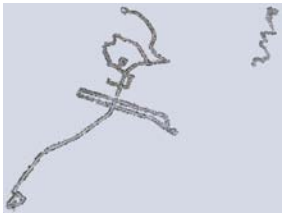
Real-world data sets: We have used our system for interactive editing and visualization of three large 3D scanner data sets (see the accompanying video for an interactive demonstration). The first is a scan of an archaeological site at Ephesus, Turkey (courtesy of M. Wimmer, TU Vienna). The data set has been acquired using a laser range scanner and consists of 14 million points. This data set is small enough to fit into main memory. Therefore, we use the in-core implementation of our hierarchical data structure for visualization. As shown in the accompanying video, we can perform interactive editing in this scene while updating the multi-resolution representation on-the-fly.

The second data set is a scan of the interior of a building acquired with a custom scanning platform [BFW*05] (courtesy of P. Biber and S. Fleck, University of Tuebingen). It consists of 75.7 million points (6.5GB of raw data). Importing the data set into the system took 85min:58sec for parsing the ASCII input file and writing it back to a temporary binary file and additional 16min:57sec sec for building the out-of-core multi-resolution hierarchy. This corresponds to an average insertion rate of 74,400 points per second. In the final hierarchy, 20% of the disk space is used by inner (multi-resolution) nodes and 80% by leaf nodes. Once the hierarchy has been constructed, interactive visualization and rendering can be performed as demonstrated in the video. As also shown in the video, we can perform local editing operations in real time on this data set. In addition, we have also created a script that performs a large scale modification of the data set, planning the operations on a $1:100$ resampled version of the data set and then executing the recorded commands offline. The test case shows removing part of the roof and clouds of outliers from the raw data set. Planning is real-time and offline processing takes 13min (removing 21.5 million points). As an example of a full-data set batch processing operation, we have computed the average point spacing from nearest neighbors, which is necessary for correct rendering. Estimating sample spacing from 9 nearest neighbors out-of-

(a) Ephesus: archaeological data set, 14M points (courtesy of M. Wimmer, TU Vienna)



(b) scan of a corridor in a building, 75M points (courtesy of P. Biber and S. Fleck, University of Tuebingen)



(c) structure-from-motion scan of an urban environment, 2.2 billion points (courtesy of J.M. Frahm, UNC)

(d) a closeup of the data set shown in (c)

Figure 4: *Example scenes - see video for details*

core took 11h for this data set. This is externally efficient (the average CPU utilization has been almost 100%), but significantly costlier than in-core or streaming out-of-core processing, as a full multi-resolution representation is kept up-to-date during processing. These additional costs are of course only warranted if interactive visualization is needed.

The third data set is a structure-from-motion reconstruction of an urban area (courtesy of J.M. Frahm, University of North Carolina [AFM*06]). The raw data set consists of 63.5GB of binary data in multiple files, providing position, normal, color and sample spacing information. Importing the data into the data structure took 14h:15min (corresponding to 43,000 points/sec) as well as additional 7h:9min for parsing the input data and storing it to a single linear binary file. Using the multi-resolution data structure, real-time walkthroughs and interactive modifications are possible as shown in the video accompanying this paper.

6. Conclusions and Future Work

We have described a new dynamic multi-resolution out-of-core data structure for rendering and editing large point clouds efficiently and a prototype editing system that uses this data structure for interactive editing of large data sets. The main contribution here is making such a data structure fully dynamic with special focus on high efficiency of the dynamic operations without sacrificing external efficiency or GPU utilization. The data structure can be updated at typical rates of 300,000 points/s for small in-core data sets and $40,000 - 70,000$ points/s for very large out-of-core data sets. The main limitation in comparison with non-dynamic multi-resolution data structures is the need to employ uniform grid quantization for resampling in inner nodes, not permitting adaptive sampling. In addition, we have described a novel large scene editing system. Given the growing amount of large data sets from 3D scanning devices, we believe that the ability for interactive editing of such data sets will become increasingly important in practical applications. In future work, several enhancements to the current data structure could be examined. We would like to implement general multi-resolution filtering schemes based on fractional weights and extend the data structure to handle dynamic multi-resolution triangle meshes based on a combination of vertex clustering with counting and edge contraction.

Acknowledgements

The authors wish to thank Peter Biber, Jan-Michael Frahm, Sven Fleck, David Gallup, Marc Pollefeys, and Michael Wimmer for providing data sets, Matthias Fisher for fruitful discussion, Prof. W. Straßer for his long term support of this project, and the anonymous reviewers for their helpful comments. This work has been supported by DFG grant "Perceptual Graphics", BW-FIT grant "Gigapixel Displays", the state of Baden Württemberg, and the Max Planck Center for Visual Computing and Communication.

References

[AFM*06] AKBARZADEH A., FRAHM J.-M., MORDOHAI P., CLIPP B., ENGELS C., GALLUP D., MERRELL P., PHELPS M., SINHA S., TALTON B., WANG L., YANG Q., STEWENIUS H., YANG R., WELCH G., TOWLES H., NISTER D., POLLEFEYS M.: Towards urban 3d reconstruction from video. In *Proc. 3DPVT '06* (2006), pp. 1–8.

[AG06] ATLAN S., GARLAND M.: Interactive multiresolution editing and display of large terrains. In *Computer Graphics Forum* (June 2006), vol. 25-2, pp. 211–224.

[BBS94] BERMAN D. F., BARTELL J. T., SALESIN D. H.: Multiresolution painting and compositing. In *Proc. SIGGRAPH '94* (1994), pp. 85–90.

[BFW*05] BIBER P., FLECK S., WAND M., STANEKER D., STRASSER W.: First experiences with a mobile platform for flexible 3d model acquisition in indoor and outdoor environments – the wägele. In *Proc. 3D-ARCH '05* (2005).

[BGB*05] BORGEAT L., GODIN G., BLAIS F., MASSICOTTE P., LAHANIER C.: Gold: interactive display of huge colored and textured models. In *Proc. SIGGRAPH '05* (2005), vol. 24-3, pp. 869–877.

[BS06] BOUBEKEUR T., SCHLICK C.: Interactive out-of-core texturing using point-sampled textures. In *Proc. SPBG '06* (August 2006).

[CGG*04] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.: Adaptive tetrapuzzles - efficient out-of-core construction and visualization of gigantic polygonal models. In *Proc. SIGGRAPH '04* (2004), vol. 23-3.

[CMRS03] CIGNONI P., MONTANI C., ROCCHINI C., SCOPIGNO R.: External memory management and simplification of huge meshes. *IEEE TVCG 9*, 4 (2003), 525–537.

[GBBK04] GUTHE M., BORODIN P., BALÁZS A., KLEIN R.: Real-time appearance preserving out-of-core rendering with shadows. In *Proc. EGSR '04*. EG Association, 2004, pp. 69–79.

[GBK04] GUTHE M., BALÁZS A., KLEIN R.: Real-time out-of-core trimmed nurbs rendering and editing. In *Proc. VMV '04* (November 2004), pp. 323–330.

[GM04] GOBBETTI E., MARTON F.: Layered point clouds. In *Proc. SPBG '04* (2004).

[GM05] GOBBETTI E., MARTON F.: Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. In *Proc. SIGGRAPH '05* (2005), pp. 878–885.

[Hop96] HOPPE H.: Progressive meshes. In *Proc. SIGGRAPH '96* (1996), pp. 99–108.

[Hop98] HOPPE H.: Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proc. VIS '98* (1998), pp. 35–42.

[IG03] ISENBURG M., GUMHOLD S.: Out-of-core compression for gigantic polygon meshes. In *Proc. SIGGRAPH '03* (2003), pp. 935–942.

[IL05] ISENBURG M., LINDSTROM P.: Streaming meshes. In *Proc. VIS '05* (2005), p. 30.

[KKF*02] KLEIN J., KROKOWSKI J., FISCHER M., WAND M., WANKA R., AUF DER HEIDE F. M.: The randomized sample tree: a data structure for interactive walkthroughs in externally stored virtual environments. In *Proc. VRST '02* (2002), pp. 137–146.

[Lin00] LINDSTROM P.: Out-of-core simplification of large polygonal models. In *Proc. SIGGRAPH '00* (2000), pp. 259–262.

[Lin03] LINDSTROM P.: Out-of-core construction and visualization of multiresolution surfaces. In *Proc. I3D '03* (2003), pp. 93–102.

[Mye98] MYERS B. A.: Scripting graphical applications by demonstration. In *Proc. CHI '98* (1998), pp. 534–541.

[Paj05] PAJAROLA R.: Stream-processing points. In *Proc. VIS '05* (2005), p. 31.

[PGK02] PAULY M., GROSS M., KOBBELT L. P.: Efficient simplification of point-sampled surfaces. In *Proc. VIS '02* (2002), pp. 163–170.

[PKG06] PAULY M., KOBBELT L. P., GROSS M.: Point-based multiscale surface representation. *ACM Trans. Graph. 25*, 2 (2006), 177–193.

[PKKG03] PAULY M., KEISER R., KOBBELT L. P., GROSS M.: Shape modeling with point-sampled geometry. In *Proc. SIGGRAPH '03* (2003), pp. 641–650.

[PZvBG00] PFISTER H., ZWICKER M., VAN BAAR J., GROSS M.: Surfels: surface elements as rendering primitives. In *Proc. SIGGRAPH '00* (2000), pp. 335–342.

[RB93] ROSSIGNAC J., BORREL P.: Multi-resolution 3d approximations for rendering complex scenes. In *Modeling in Computer Graphics* (1993), Springer, pp. 455–465.

[RL00] RUSINKIEWICZ S., LEVOY M.: Qsplat: a multiresolution point rendering system for large meshes. In *Proc. SIGGRAPH '00* (2000), pp. 343–352.

[RL01] RUSINKIEWICZ S., LEVOY M.: Streaming qsplat: a viewer for networked visualization of large, dense models. In *Proc. I3D '01* (2001), pp. 63–68.

[Sam90] SAMET H.: *The design and analysis of spatial data structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[SG01] SHAFFER E., GARLAND M.: Efficient adaptive simplification of massive meshes. In *Proc. VIS '01* (2001), pp. 127–134.

[SG05] SHAFFER E., GARLAND M.: A multiresolution representation for massive meshes. *IEEE TVCG 11*, 2 (2005), 139–148.

[Wan04] WAND M.: *Point-Based Multi-Resolution Rendering*. PhD thesis, University of Tuebingen, 2004.

[WFP*01] WAND M., FISCHER M., PETER I., AUF DER HEIDE F. M., STRASSER W.: The randomized z-buffer algorithm: interactive rendering of highly complex scenes. In *Proc. SIGGRAPH '01* (2001), pp. 361–370.

[WH03] WLOKA M., HUDDY R.: Directx 9 performance: Where does it come from, and where does it all go? game developers conference 2003 presentation, 2003.

[WPK*04] WEYRICH T., PAULY M., KEISER R., HEINZLE S., SCANDELLA S., GROSS M.: Post-processing of scanned 3d surface data. In *Proc. SPBG '04* (2004).

[WS06] WIMMER M., SCHEIBLAUER C.: Instant points. In *Proc. SPBG '06* (2006), pp. 129–136.

[XGR07] The xgrt system. http://www.gris.uni-tuebingen.de/people/staff/pjenke/xgrt/, 2007.

[YSGM04] YOON S.-E., SALOMON B., GAYLE R., MANOCHA D.: Quick-vdr: Interactive view-dependent rendering of massive models. In *Proc.VIS '04* (2004), pp. 131–138.

[ZPKG02] ZWICKER M., PAULY M., KNOLL O., GROSS M.: Pointshop 3d: an interactive system for point-based surface editing. In *Proc. SIGGRAPH '02* (2002), pp. 322–329.

[ZSS97] ZORIN D., SCHRÖDER P., SWELDENS W.: Interactive multiresolution mesh editing. In *Proc. SIGGRAPH '97* (1997), pp. 259–268.