FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

External Sorting Of Point Clouds

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor Of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Kurt Leimer
Matrikelnummer 0825842

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Michael Wimmer
Mitwirkung: Claus Scheiblauer

Wien, TT.MM.JJJJ

(Unterschrift Verfasser)

(Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

External Sorting Of Point Clouds

BACHELOR THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor Of Science

in

Media Informatics and Visual Computing

by

Kurt Leimer
Registration Number 0825842

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Michael Wimmer
Assistance: Claus Scheiblauer

Vienna, TT.MM.JJJJ

(Signature of Author) (Signature of Advisor)

Erklaerung zur Verfassung der Arbeit

Kurt Leimer
Brigittaplatz 1/4/2, 1200 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Abstract

In recent years, points have seen increased use as a rendering primitive. This Bachelor Thesis presents and compares a number of sorting algorithms used for sorting points in a preprocessing step. This is done in order to decrease the time needed to create a point cloud model that can be rendered by the GPU in real time. Points are compared either by their position along the longest axis or by Morton order and sorted using heap sort or radix sort. An external merge sort algorithm is used for large datasets that do not completely fit into memory. The speed of the sorting process is furthermore increased by making use of parallel processing.

Kurzfassung

In den letzten Jahren stieg die Verwendung von Punkten als Rendering-Primitiv stark an. Diese Bachelorarbeit präsentiert und vergleicht einige Sortier-Algorithmen, die dazu genutzt werden, die Punkte in einem Vorverarbeitungs-Schritt zu sortieren. Dadurch kann die Zeit verringert werden, die benötigt wird, um ein Punktwolken-Model zu erstellen, das von der GPU in Echtzeit gerendert werden kann. Punkte werden entweder nach ihrer Position entlang der längsten Achse oder per Morton Order verglichen und mithilfe von Heap Sort oder Radix Sort sortiert. Ein externer Merge Sort-Algorithmus wird für Datensätze verwendet, die nicht komplett in den Arbeitsspeicher passen. Die Geschwindigkeit des Sortier-Prozesses wird außerdem mithilfe von paralleler Verarbeitung beschleunigt.

Contents

CHAPTER 1

Introduction

A point cloud is a set of multiple points used for rendering. A point in the mathematical sense is a zero-dimensional entity in space. Its only attribute is its location where it occupies an infinitesimal amount of space and therefore it is not visible. When talking about a point as a rendering primitive, we assume that its location is visualized as at least one pixel on screen when visible. Points may also have a variety of additional attributes, like color, opacity and a normal vector (if it is part of a surface). Point data can be acquired in a variety of ways. Levoy and Whitted [9] have shown that all geometrical models can be converted to points.

The data can also be obtained directly, for example by means of a range scanner or scientific datasets. A range scanner takes discrete distance measurement samples of a real-world environment, resulting in a set of 3D coordinates. The scanner rotates in discrete angles around its vertical axis and takes samples at each angle along a vertical line. A distance sample is taken by measuring the time of flight of a laser pulse. Additionally, images can be taken from the environment by using a camera mounted on top of the scanner. These images can later be applied to the distance samples, resulting in a colored point cloud.

To scan a whole environment, multiple scans from various positions might be necessary, as objects may occlude parts of the environment from one scan position. The samples that have been obtained by one scan are all stored in a file. When using multiple scan positions, this results in a number of files containing the point data that later need to be combined for rendering. As each file contains all the samples from a single scanning position, combining the files results in a rather arbitrary sorting of point samples. A simple example of a scanning environment is illustrated in Figure 1.1: The small dark circles show the positions of the scanners, the larger bright circles illustrate the range of each scanner and the black rectangles are objects that are part of the environment. To capture each side of the objects, scans from multiple positions are necessary (though only 3 are shown for simplicity).

To make rendering in real-time possible, the points must first be stored in a fitting data structure. When working with datasets that are larger than main memory, this build up process can take several hours, as parts of the data structure need to be written to and loaded from the

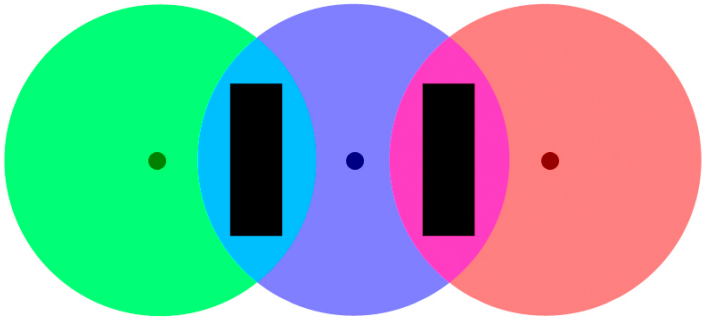

Figure 1.1: A simple sketch of a scanning environment (top-down view). The small darker circles show the positions of the scanners, the large brighter circles illustrate the range of each scanner and the black rectangles are objects that are part of the environment.

hard disk dynamically. This is especially true when the input data is unsorted, as is often the case with data obtained by a range scanner.

Taking Figure 1.1 as an example, the unsorted point data of the environment may be processed in the order green-red-blue. If not all points can be stored in memory at the same time, the part of the data structure containing the points of the first file needs to be written to the hard disk when processing the second file, and needs to be loaded again when processing the third file. To avoid this, the ideal situation would be if the points were sorted according to their spatial proximity. This way, the point samples of the object on the left can be processed completely before processing the point samples of the object on the right.

The aim of this Bachelor Thesis is to speed up the build up process by sorting the unprocessed point data in a preprocessing step, either by axis or by Morton order. The sorting algorithms were implemented as a Qt-Plugin for the C++ application Scanopy, a point cloud renderer and editor developed by the Institute of Computer Graphics and Algorithms at the Vienna University of Technology. Since the application uses a Modifiable Nested Octree [15] as a data structure, the following Chapter will give an explanation of the data structure and an overview of the previous work it was based on. Chapter 3 gives a description of the sorting algorithms that were used, while Chapter 4 contains the achieved results.

CHAPTER 2

Previous Work

To make real-time rendering of point clouds possible, the points first need to be sorted into a fitting data structure. Furthermore, point cloud models are often larger than the space available in main memory, so certain techniques need to be employed to dynamically load parts of the point cloud into memory during rendering. Subchapter 2.1 describes some data structures for fast point cloud rendering, while Subchapter 2.2 deals with the out-of-core rendering problem.

2.1 Fast Point Cloud Rendering

Fast point rendering algorithms sacrifice visual quality in favor of rendering speed. Since the GPU that is in charge of rendering needs quick access to the points, it is necessary to store the points in a data structure that can be handled by the GPU. Examples of previous work in the field include QSplat [13], where a hierarchy of bounding spheres is used, and ρ-Grids [4], a hierarchy of recursively subdivided grids in which points are stored. The data structure that is used in Scanopy is called Memory Optimized Sequential Point Tree [14], which is based on the Sequential Point Tree [3] data structure.

Similarly to QSplat and ρ-Grids, Sequential Point Trees (SPT) use a hierarchical data structure in which points are stored. Points are first sorted into an octree. Each leaf node of the octree stores one of the original points, a normal vector and a bounding sphere diameter. The inner nodes of the octree contain the averaged position and normal values of their child nodes, as well as the diameter of a bounding sphere that contains all bounding spheres of its children. During rendering the octree is traversed depth-first. For each node, an image error is calculated, and if the error is above a certain threshold, the algorithm steps down one level. Otherwise the node is rendered.

The octree needs to be sequentialized so it can be processed by the GPU, therefore a different error metric is necessary. By using the calculated image error it is possible to calculate a minimum and maximum distance for each node. When the distance between the node and the viewpoint lies between the calculated minimum and maximum distance, the node is rendered,

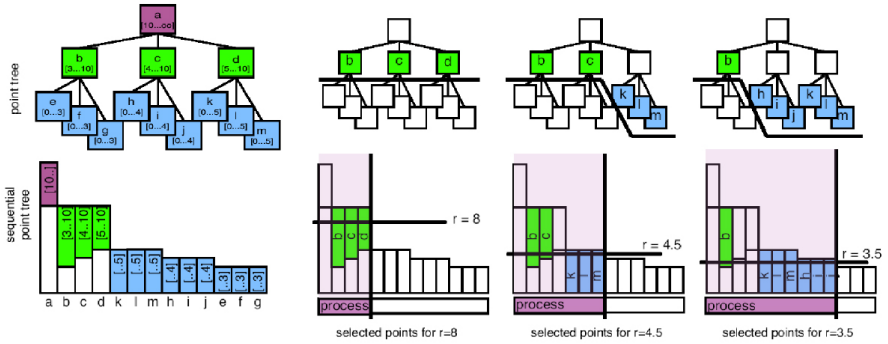

Figure 2.1: Hierarchical and sequential representation of a SPT.

otherwise it is skipped. By sorting the nodes according to their maximum distance, all nodes whose maximum distance is too small to be rendered for the current viewpoint can be culled before sending the sequential list of nodes to the GPU. After that, the GPU only needs to skip the nodes whose minimum distance is too large to be rendered. Figure 2.1 shows examples of both the hierarchical and sequential representation of SPTs.

The main idea behind Memory Optimized Sequential Point Trees (MOSPT) is to lessen the memory requirements of SPTs. In contrast to the latter, MOSPTs only need positional information for each point - normals and bounding sphere diameters are not required. This is especially useful for data sets which do not contain additional information, for example data created by range scanners.

For model build-up, points are first sorted into an octree. Then, for each inner node, one of the points in the child nodes is pulled up (Figure 2.2). The point in question is chosen by averaging the information of the child nodes and selecting the point that is closest to that value. By using this approach, each inner node contains one of the original points, therefore saving space because no additional points with averaged information are created.

To select a level of detail (LOD) for rendering, the bounding sphere diameter of the closest vertex in the node's bounding box is projected to the screen. If its length is above a certain threshold, the next level in the MOSPT is examined. If it is below, then the appropriate LOD has been found. One disadvantage of MOSPTs is that only one LOD can be selected for the whole model because the bounding sphere diameter is not stored with the points. Therefore, more points than necessary might be rendered.

Like SPTs, MOSPTs need to be sequentialized so they can be processed by the GPU. The points are sorted breadth-first, so the root node is always first. Since only one LOD level is rendered, only the nodes of the selected LOD and below need to be rendered.

2.2 Out-of-Core Point Cloud Rendering

Another problem that needs to be dealt with is large point clouds that do not fit into main memory. A variety of suggested solutions already exist for this problem. With Layered Point

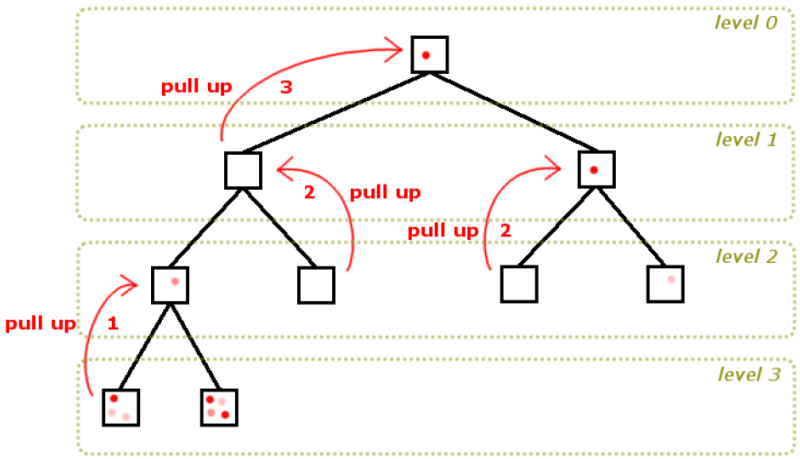

Figure 2.2: In a MOSPT, instead of creating a new point with the averaged information of its children, one of the child points is pulled up.

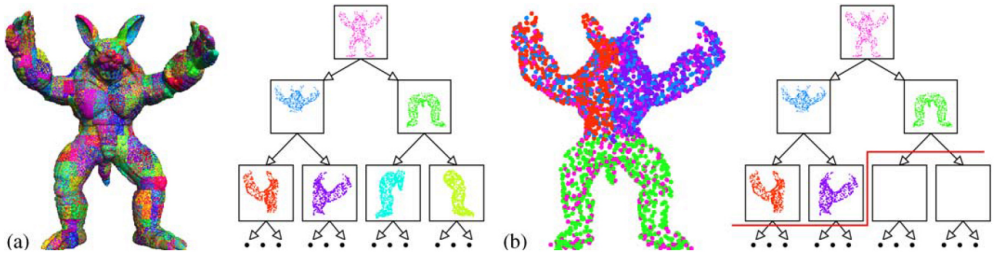

Figure 2.3: A point cloud model consisting of Layered Point Clouds. The points of the model are colored based on the node they are stored in. (a) The whole model and its hierarchical representation. (b) The model as rendered with a certain level of detail (indicated with a red line in the hierarchical representation).

Clouds [6], only a subset of the total set of points is loaded when the user views the model from far away. When the user zooms in on a certain part of the model, more points are loaded for this part of the model, decreasing the average distance between the points. This can be achieved by recursively subdividing the model into parts. The resulting hierarchy of the model can be portrayed as a tree, as can be seen in Figure 2.3. For each node, starting with the root node, a subset of points is chosen from the total set of points to create a point cloud representing the corresponding part of the model. During rendering, the number of point clouds that is loaded for a specific part of the model depends on the view distance. Thus, the details of the model are further refined the closer the viewpoint is.

With XSplat [12], points are first sorted into an octree. The nodes of the tree then get assigned a layer number, which corresponds to the longest path between the node and its leaf nodes. A sequential representation is generated by sorting all nodes according to their layer numbers, while those nodes whose layer number is the same are sorted breadth-first. Finally,

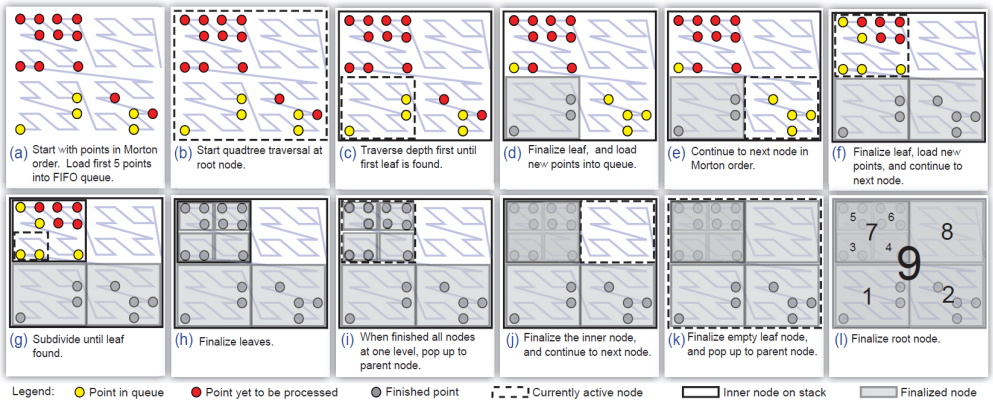

Figure 2.4: A 2D example of the octree construction algorithm by Kontkanen et al.

the points in the nodes are divided into blocks of a certain size, starting with the first node that is larger than the block size. Some additional information is stored with each block, such as the bounding sphere radius or the averaged attributes of the block's points. This information can then be used during rendering to decide whether or not a block should be loaded and whether its points should drawn on screen.

Wand et al. [17] also use an octree for their multi-resolution data structure. The space occupied by the inner nodes of the octree is divided with a quantization grid, where each grid cell contains a representative point of the space it occupies. The rest of the points are stored in the leaf nodes of the octree. During rendering, nodes of the octree can be loaded dynamically depending on the level of detail that is necessary at any given time. One advantage of the data structure is that it can be edited dynamically by inserting or deleting points. When inserting a point causes a leaf node to exceed its capacity of points, the points stored in the node are moved into newly created leaf nodes, while the original leaf node becomes a new inner node. When deleting a point from a leaf, the points stored inside the leaf and its siblings can be merged and moved to their parent node if the number of points is smaller than the maximum capacity of a leaf node.

Kontkanen et al. [8] present an efficient out-of-core algorithm for sorting points into an octree. Using an external merge sort algorithm (see Subchapter 3.5), points are first sorted according to their Morton order values, which correspond to their position along a depth-first traversal of an octree (see Subchapter 3.2). Also, it is necessary to decide on the maximum number of points *leaf_max* that can be stored in a leaf node of the octree. Figure 2.4 shows a 2D example of the construction process with *leaf_max* = 4. To begin, the first *leaf_max*+1 points are loaded and held in memory. Then the algorithm checks whether the current node of the octree (starting with the root node) can store all of the loaded points that fall into the space occupied by the node. If this is not the case, the algorithm steps down a level in the octree hierarchy. Otherwise, the corresponding points are stored in the node and the node is finalized by calculating the node's attributes based on the attributes of the points stored inside the node. Then more points are loaded so the number of points held in memory is always *leaf_max*+1 (unless there are no more points to load). Inner nodes are finalized once all of their child nodes

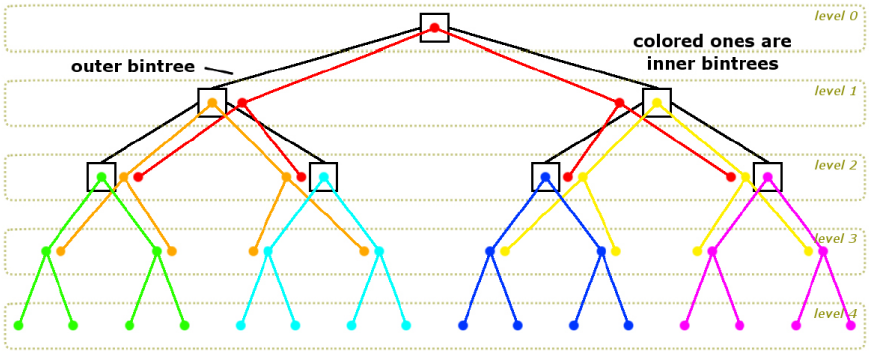

Figure 2.5: Each node of the Nested Octree contains a MOSPT.

are finalized, resulting in a depth-first construction of the octree.

Marek [10] proposes a way to externally sort point clouds using AVL-Trees. An AVL-Tree is a self-balancing binary search tree, which means that the ordering of nodes in the tree hierarchy is adjusted any time a node is inserted or deleted, so that the heights of the subtrees of a node differ by no more than 1. This ensures a worst and average case complexity of O(log n) for search, insertion and deletion operations, with n being the number of nodes in the tree. To sort a point cloud along an axis, the points are distributed into buckets corresponding to their position on the axis. These buckets are stored in the leaf nodes of the AVL-Tree and have a maximum number of points they can hold. If a point is inserted into a full bucket, the corresponding leaf node is split into 2 new leaf nodes, each containing half of the points of the bucket. Since not all points can be held in memory at the same time, a least-recently-used (LRU) cache is used. When the cache is full, the bucket that has been used least recently is written to the hard disk and is only retrieved when it is needed again. Once all points have been distributed into the buckets, each bucket can be sorted individually using a non-external sorting algorithm.

Scanopy uses a Modifiable Nested Octree [15] for dealing with the out-of-core rendering problem, which is based on the Nested Octree [14] data structure. Nested Octrees provide an outer hierarchy where at each node a MOSPT is stored. This causes the MOSPTs to overlap - for example, the points stored in the second level of the root node's MOSPT share the same space as the root node of the MOSPTs stored in the second level of the Nested Octree. This can be seen in Figure 2.5. All MOSPTs have the same number of levels, which also corresponds to the maximum number of points that share the same space. The number of levels must be selected carefully - a too small number results in too many files on disk, while a too large number may result in more points being rendered than necessary. Figure 2.6 shows a two-dimensional example of the points stored at various levels of a Nested Quadtree. Nested Octrees also enable an enhanced LOD selection and the possibilities of view-frustum culling.

A Nested Octree is constructed beginning with the creation of a MOSPT for the root node, which shares the same space as the bounding box for the whole model. The points of the input file for that particular node are inserted into the MOSPT. When a leaf node of the MOSPT already contains a point, any points that would be inserted into the same leaf node are rejected

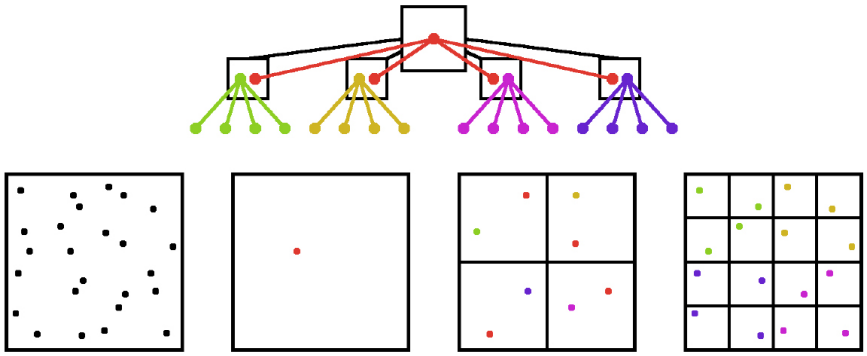

Figure 2.6: The points stored at various levels of a Nested Quadtree.

and written into a rejection file with an identifier corresponding to the node. Once all points of the input file have been processed, the points in the rejection files may be added back into the corresponding leaf nodes, but only if their amount is below a certain threshold. Afterward, points from the leaf nodes are pulled up into the parent nodes and the MOSPT is written to the hard disk. Then, a new child node of the current node is created and if a rejection file exists for this node, the whole process is repeated for this node. At the end, the Nested Octree itself is written to the hard disk.

The Nested Octree is small enough to fit into main memory at all times during rendering. The MOSPTs within can be loaded dynamically. If a node is not within the view-frustum it is skipped. Nodes whose points are too far away so that their projection on screen is too small to contribute to the appearance are skipped as well. LOD selection is also improved, because a different LOD can be chosen for different MOSPTs.

The Modifiable Nested Octree (MNO) is an optimization of Nested Octrees. The inner hierarchy of each node is abandoned and replaced with a regular grid that stores the points of the lowest level of the MOSPT (Figure 2.7). In other words, only the leaf nodes of the MOSPT remain, which simplifies the insertion of the points. When inserting a point, the bounding box of the MNO's root node is subdivided into a regular grid. Based on the point's coordinates, it falls into one of the grid's cells. If the cell is empty, the point is stored there. Otherwise, it is inserted into the correct child node, which is calculated using the point's coordinates and the center of the parent node. This process is repeated recursively until the point falls into an empty cell. To prevent the creation of child nodes that end up mostly empty, an optimization is to hold back points that would fall into a child node that does not exist yet and only create the child node once there are multiple points that would fall into it.

To allow the build-up of models that are too large to fit into main memory, a LRU cache is used to swap nodes in and out of memory. Nodes that have been created are held in the cache to allow further points to be inserted. Once the cache is full and more nodes need to be created, the nodes that were used least recently are written to disk. If a point needs to be inserted into a node that has already been dropped, it is loaded into memory again. If this happens often,

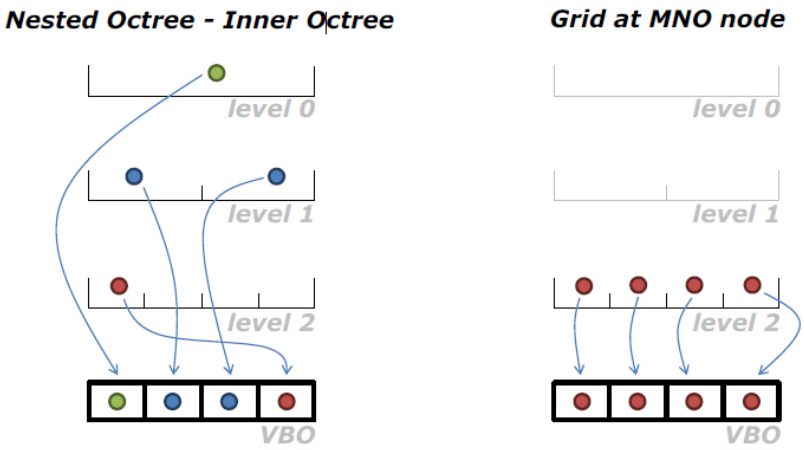

Figure 2.7: A comparison between a Nested Octree and a MNO.

the build-up process is slowed down considerably. As such, the main goal of this thesis is to minimize the amount nodes that are swapped back to memory, by sorting the input set of points in a preprocessing step.

CHAPTER 3

External Sorting

In order to find the most efficient way of sorting unprocessed point clouds and increase the speed of the model build up process as much as possible, a variety of algorithms were implemented and tested. Points are sorted either by axis or by Morton order, using either heap sort or radix sort. For large point clouds that do not fit into main memory, an external merge sort algorithm is used. The speed of the sorting process can be increased further by making use of multi-core CPU's parallel processing capabilities.

3.1 Sorting By Axis

When sorting by axis, the points are simply sorted according to their position along one of the three axes in Euclidean space. Early tests have shown that the best result is achieved when sorting along the longest axis and also comparing the points by their other coordinates if their position along the longest axis is the same. The length of each axis can easily be retrieved from the bounding box of the point cloud.

3.2 Morton Order

By using Morton order [11] it is possible to map multidimensional data to one dimension, while preserving locality between data points. Sorting points by Morton order yields the same result as a depth-first traversal of an octree, which would suggest that using Morton order for sorting point clouds prior to storing the points in an octree might produce good results. Connecting the ordered points yields a Z-order curve, a type of space-filling curve which is named after its shape, as can be seen in Figure 3.1.

Points are sorted according to their Z-order curve value (Z-value). When using integers, the Z-value can be calculated by bit-interleaving the coordinates of the point, as can be seen in Figure 3.2. As bit-interleaving can be a very time-consuming operation, especially for large and high-dimensional data sets, a faster way to compare two points by Morton order was presented

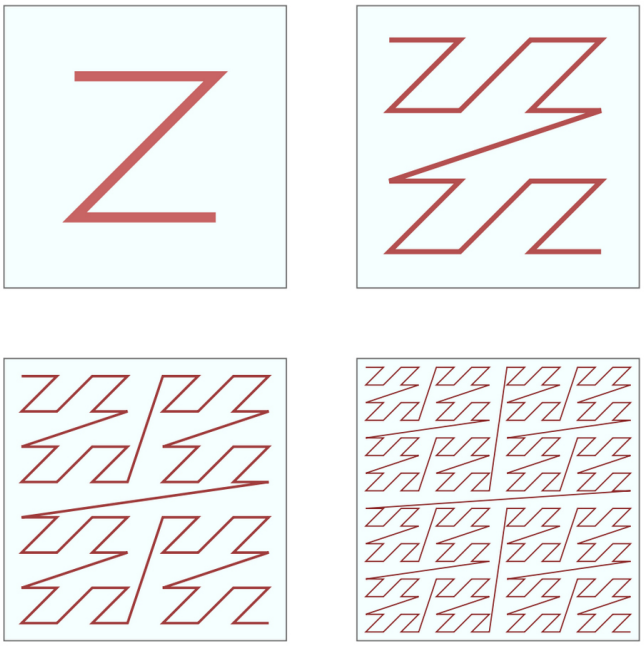

Figure 3.1: Four iterations of a Z-order curve.

by Chan [1]. For each dimension, the bit-representation of the coordinates of the two points are compared with the XOR bit-operation. By identifying the dimension with the most significant differing bit (MSDB), the points need only be compared by this coordinate with the less-than-operator . Algorithm 3.1 shows the pseudo-code for this operation.

It is not possible however to directly apply this algorithm when using floating point values. C++ floating-point numbers use the IEEE Standard for Floating-Point Arithmetic format [5].

The first bit is the sign bit. If it is set, the value is negative, otherwise it is positive. The following 8 bits store the exponent (which is biased by 127, so the actual range of the exponent is -127 to 128) and the remaining 23 bits store the mantissa so that the floating point value can be computed with $(-1)^{sign} * 2^{exponent-bias} * (1 + 2^{-23} * mantissa)$.

A possible algorithm supporting floating point numbers was presented by Connor and Kumar [2] and can be seen in Algorithm 3.2. In order to find the dimension by which the two points have to be compared, the exponents of the coordinates are examined first. If the two exponents are different for every dimension, the points are compared by the dimension in which either point has the highest exponent, as the exponent can be considered the equivalent of the MSDB when comparing points with integer value coordinates.

When the exponents are equal for a dimension, the mantissas have to be factored in. This is done by first XORing the mantissas, then converting the result to a floating point value and adding its exponent (which is always negative) to the original exponent.

The only thing that remains is dealing with negative numbers. When treating the whole 3-dimensional space as the root node of an octree, every octant is one of its child nodes. Therefore,

Algorithm 3.1: Integer Morton Order Algorithm

```
   Data: n-dimensional points p and q
   Result: true if p < q in Morton order
 1 procedure COMPARE (point p, point q)
 2     x ← 0; dim ← 0;
 3     for i = 0 to n do
 4         y ← p_(i) XOR q_(i);
 5         if LESS_MSB(x, y) then
 6             dim ← i; x ← y;
 7         end
 8     end
 9     return p_(dim) − q_(dim)
10 end
11 procedure LESS_MSB (int x, int y)
12     return x < y AND x < (x XOR y)
13 end
```

Algorithm 3.2: Floating Point Morton Order Algorithm

```
   Data: n-dimensional points p and q
   Result: true if p < q in Morton order
 1 procedure COMPARE (point p, point q)
 2     x ← 0; dim ← 0;
 3     for i = 0 to n do
 4         y ← XOR_MSB(p_(i), q_(i));
 5         if x < y then
 6             x ← y; dim ← j;
 7         end
 8     end
 9 end
10 procedure XOR_MSB (double a, double b)
11     x ← EXPONENT(a); y ← EXPONENT(b);
12     if x = y then
13         z ← MSDB(MANTISSA(a), MANTISSA(b));
14         x ← x − z;
15         return x
16     end
17     if y < x then return x;
18     else return y;
19 end
```

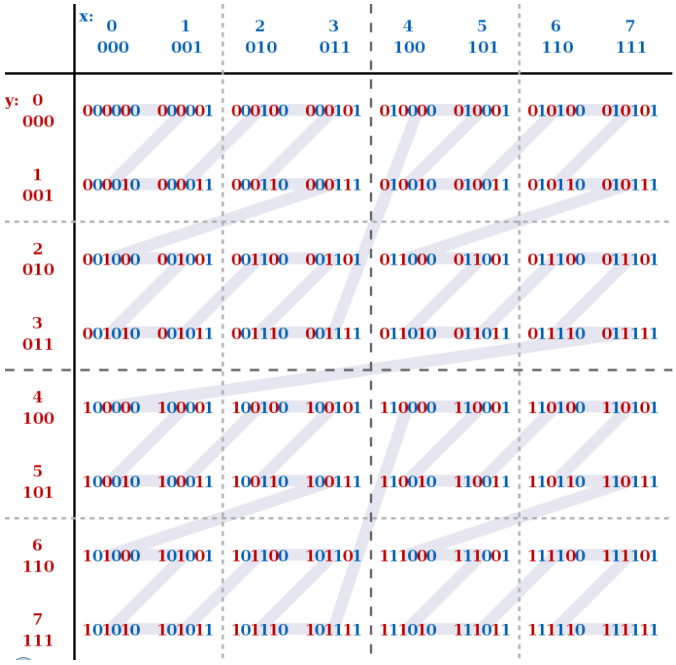

Figure 3.2: Creating a Morton order by bit-interleaving the point coordinates.

when the signs of the two point coordinates are different for at least one dimension, there is no need to examine the exponents. If the signs are the same for every dimension, they can simply be ignored when examining the exponents.

3.3 Heap Sort

As its name suggests, heap sort uses a heap data structure for sorting. A heap (Figure 3.3) is a tree with the special property that the value of every node is always larger (or smaller, if it is a min-heap) than the value of its child nodes. To construct a heap, each node is inserted at the bottom level and then compared to its parent node. If the parent node's value is smaller, then the two nodes are swapped. This process is repeated until the parent node's value is bigger or when the inserted node becomes the root node.

When deleting the root node from the heap, the last element in the heap takes its place. The new root node is then compared to its children - if they are in the correct order, the process is finished, otherwise the root node is swapped with the larger child in a max-heap, or the smaller child in a min-heap, until the order is correct.

To sort a set of values using heap sort, a heap containing all values is constructed. Then the root node is deleted from the heap and its value is inserted into the array containing the sorted values. After restoring the heap property, the process is repeated until all values have been sorted. Heap sort is a comparison-based algorithm, so it works both when comparing points by axis and Morton order, as the Morton order comparison algorithm directly compares two float

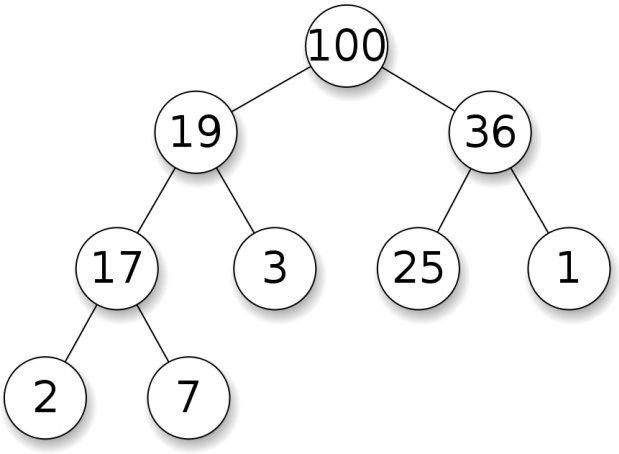

Figure 3.3: Example of a max-heap. The value of each node is larger than the value of its child nodes.

values. The worst and average case complexity of heap sort is O(N*log(N)), where N is the number of elements that have to be sorted.

3.4 Radix Sort

Radix sort is a non-comparative sorting algorithm that is primarily used for integers. In each pass, values are grouped according to a single individual digit, starting with either the least significant or the most significant digit. The result of the pass is then sorted according to the next digit and so on. An example can be seen in Figure 3.4. This can also be applied to the binary representation of types other than integer, making sorting of floating point values with radix sort possible, as described by Terdiman [16] - because both the exponent and the mantissa of floating point values are always positive, and a value with a larger exponent is always larger regardless of the mantissa, floating point values can be sorted according to their individual bytes, or even multiple bytes at once. In our case, the points are grouped according to 2 bytes at once.

The only problem remaining is the sign bit. When it is set, the binary representation of a negative value would always be larger than that of a positive value. An efficient way of dealing with negative numbers was presented by Herf [7]. Flipping the sign bit for every value results in positive values being larger than negative values. However, that alone is not enough, as the numbers are signed-magnitude, which means that the binary representation of a negative value will be larger the 'more negative' it is. The solution is to not only flip the sign bit, but all other bits as well for negative numbers, which can be done at low cost by XORing the value with a mask where every bit is set.

At the beginning of the sorting process, the offsets for a temporary array (which is of the same size as the array in which the original values are stored) are calculated by counting the number of floating point numbers with the same byte values. These offsets correspond to the

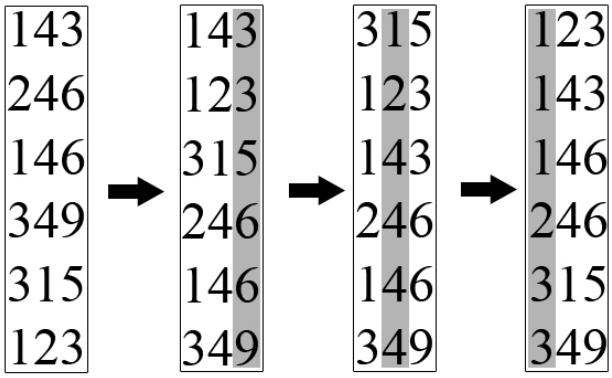

Figure 3.4: An example of radix sort. The values are first grouped according to the last digit, then the middle digit, and finally the first digit.

size of the buckets into which the floats are sorted according to their byte values. Each floating point number gets inserted into the temporary array at the offset pertaining to its value of the bytes currently examined. Once all numbers have been inserted, the process is repeated with the next 2 bytes and the numbers are swapped back to the original array. In the case that all values of a single byte position are the same for every number, the sorting pass is skipped to save time, as the sorting pass would be useless.

Since radix sort is a non-comparative sorting algorithm, it cannot be used with our Morton order comparison. As mention earlier, for sorting points by axis, it is necessary to sort along more than one axis. Since radix sort is temporally coherent, it is possible to use the algorithm on the point cloud once for each axis, beginning with the shortest.

The worst and average case complexity of radix sort is O(N*k), where N is the number of elements and k the number of passes. One pass is needed to calculate the offsets and 2 passes are needed to sort the points according to their byte values. This is done once for each axis, so the total complexity is O(9N), which is significantly lower than the complexity of heap sort which is O(N*log(N)). The biggest disadvantage of radix sort is that it needs a temporary array to store the sorted values, therefore it needs more memory than heap sort.

3.5 External Merge Sort

In order to deal with large data sets that do not completely fit into main memory, an out-of-core sorting algorithm has to be used. External merge sort is a comparative sorting algorithm with 2 steps. In the first step, the data is split into multiple chunks that each fit into main memory. These chunks are sorted one by one (for example by using heap sort or radix sort as previously described) and then saved to the hard drive.

In the second step, the chunks are merged. The first element of each chunk is read and then they are compared to each other. The smallest element is stored in the sorted array and the next

element of the chunk is read and compared to the others. Since all chunks have already been sorted, the end result will have all points in the correct order.

When there are only 2 chunks, the merge process is simple, as it is easy to compare 2 elements and determine from which chunk the next element has to be read. Once all points of one chunk have been stored in the sorted array, the rest of the points in the other chunk can be stored in the array as they are, since they are already sorted.

If there are more than 2 chunks, the merge algorithm uses a priority queue, like a heap. The first element of each chunk is inserted into a heap. The root node is then removed and stored in the sorted array. To read the next element, it is necessary to know which chunk the point came from, so this information has to be stored, for example as an additional attribute of the point. The complexity of a merge with priority queue is O(N*log(k)), where N is the number of elements to be merged and k is the number of chunks these elements are stored in.

3.6 Parallel Sorting

In order to increase the speed of the sorting process, the external merge sort algorithm can be adapted for use with multiple and multi-core CPUs. In the first step of the external merge sort, the data is split into chunks that fit into main memory and then those chunks are sorted one by one. Since the complexity of heap sort is O(N*log(N)), it makes sense to decrease the size of each chunk and sort multiple chunks simultaneously by using several threads. Of course this results in a slight decrease in speed during the merge process, since there are more chunks that need to be merged. However, given the difference in size between the huge number of elements that need to be sorted when dealing with point clouds and the relatively small number of chunks, the tradeoff pays off as can be seen in the results (Chapter 4).

Parallel processing can also be applied to the merge itself using a divide-and-conquer approach. By using threads to perform multiple smaller merges simultaneously, and continuing to merge the results until only one file containing the complete sorted data remains, a further increase in speed can be accomplished when using Morton order (see Chapter 4).

CHAPTER 4

Results

For testing we used a machine with a 2,40GHz Q6600 Intel Core 2 Quad Processor, 8GB RAM and a 3TB Seagate Barracuda hard drive with 7200RPM. The tests were performed on a model depicting the Vienna Stephansdom, consisting of many smaller point clouds that were scanned with a range scanner at various positions. The model contains a total of 675.344.552 points, which corresponds to roughly 10GB of data . The point cloud as rendered by Scanopy is depicted in Figure 4.1.

The size of the chunks was limited to a maximum of 50% of the memory size, so a total of 4GB of memory was available for sorting. The size of the LRU cache used during the build up of the model also has a huge impact on performance. All tests were performed twice - once with a small cache that can hold a maximum of 10 million Points (160MB) and once with a large cache whose size is 30% of the computers memory (2,4GB). When using a small cache, only few nodes of the MNO can be held in memory at the same time. Depending on how the points are sorted, this may lead to nodes being written to and loaded from the hard disk frequently, as points may need to be stored in a node that has already been written to the hard disk. These frequent disk accesses increase the time it takes to build up the model, which is also reflected in our results, the entirety of which can be seen in Table 4.1.

4.1 Build-Up Times

Using points sorted by Morton order results in a significant decrease in build-up time, regardless of the size of the LRU cache. As mentioned in the previous Chapter, Morton order is the same order one would get from a depth-first traversal of an octree. Therefore, inserting points sorted by Morton order into an MNO results in filling up the MNO depth-first. Because of this, nodes in the lowest levels of the MNO never need to be swapped back to memory because they are filled first and only written to disk once they have been filled with all points that would fall into these particular nodes. The depth-first order has another advantage - because it is ensured that all children of a node are recursively filled with points before another node of the same level is

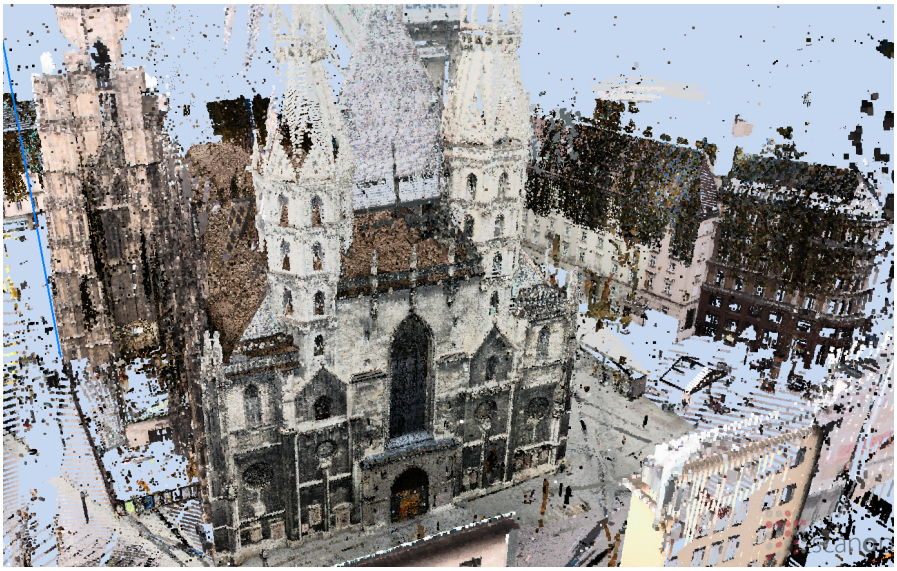

Figure 4.1: A point cloud model of the Vienna Stephansdom.

created, less nodes need to be held in memory, which further reduces the amount of nodes that need to be swapped back to memory. This makes using an input set of points sorted by Morton order very efficient during build-up even with a small cache.

The efficiency of sorting points by axis largely depends on the size of the model and cache. Because each 'slice' of points along the longest axis may contain points in up to 4 octants of space, the amount of nodes that need to be held in memory is larger than when using Morton order. If the cache is large enough, this is not a problem - because the nodes that have already been filled with all points belonging to those particular nodes are written to disk first, this results in very few nodes that have to be swapped back to memory, making the build-up process much more efficient than when using unsorted points. If the cache cannot hold enough nodes simultaneously however, this variant may become even worse than using unsorted points, as nodes need to be written to and read from the hard disk frequently.

4.2 Sorting Times

While using sorted points for building up the model is more efficient than using unsorted points in most cases, the time it takes to sort the points needs to be factored in to decide which method is the best. A variety of combinations was tested. Sorting the individual chunks by axis can be done either with heap sort or radix sort, and testing has shown that radix sort performs faster than heap sort even though the radix sort is performed once for each axis. Since radix sort produces more chunks as it needs more memory for sorting, the merge process will take longer compared to using heap sort. This is especially true for smaller models because the amount of chunks may decide whether a priority queue is needed for merging. For large models however, the gain when sorting the individual chunks is greater than the loss when merging. Parallel processing

can be used to sort multiple chunks in parallel. For both heap sort and radix sort this results in a better overall time, because more time is gained during sorting the chunks than is lost during the merge. Using a parallel merge however results in a much slower merge process because the hard disk proves to be a bottleneck - the short comparison operation leads to frequent disk accesses when multiple threads are loading the next point from disk.

The Morton order comparison operation for floating point values takes significantly longer than simply comparing them by their position along the axis. Without using multi-threading, sorting by Morton order takes more than 10 times as long as sorting by axis using heap sort. The sorting time can be reduced by a large margin using parallel processing however - since the comparison operation is so costly, too frequent disk accesses are not a problem. Using parallel sorting of chunks and parallel merge results in a sorting time decrease of 68%, which is still slower than both radix or heap sorting by axis however.

4.3 Overall Performance

In most cases, both sorting by axis and sorting by Morton order before building up the model results in a significant speed up when compared to using unsorted points. Sorting by axis when using a small cache during the model build-up is the only exception - if the amount of data is large and the density of points high, the build-up process might take even longer than when using unsorted points. If this is the case, sorting by Morton order has a clear advantage over sorting by axis, because even though the sorting process takes longer, the overall time is the shortest compared to using unsorted points or points sorted by axis. If the cache is large enough however, sorting by axis is the fastest way, as the speed gain during the build-up process is about the same as when using Morton order, but the sorting process is significantly faster.

Large Build-Up Cache (2,4GB)			
Sorting Type	***Sorting Time***	***Build-Up Time***	***Total Time***
Unsorted	-	34h 15m 28s	34h 15m 28s
Axis	HSS: 21m 01s	38m 43s	59m 44s
	HPS: 17m 05s		55m 48s
	HSP: 24m 59s		1h 03m 42s
	HPP: 22m 13s		1h 00m 56s
	RSS: 14m 53s		53m 36s
	RPS: 13m 23s		52m 06s
	RSP: 19m 48s		58m 31s
	RPP: 22m 39s		1h 01m 22s
Morton Order	HSS: 4h 46m 12s	39m 45s	5h 25m 57s
	HPS: 2h 10m 45s		2h 50m 30s
	HSP: 4h 35m 21s		5h 15m 06s
	HPP: 1h 31m 20s		2h 11m 05s

Small Build-Up Cache (160MB)			
Sorting Type	***Sorting Time***	***Build-Up Time***	***Total Time***
Unsorted	-	50h 35m 08s	50h 35m 08s
Axis	HSS: 21m 01s	60h 39m 45s	61h 00m 46s
	HPS: 17m 05s		60h 56m 50s
	HSP: 24m 59s		61h 04m 44s
	HPP: 22m 13s		61h 01m 58s
	RSS: 14m 53s		60h 54m 38s
	RPS: 13m 23s		60h 53m 08s
	RSP: 19m 48s		60h 59m 33s
	RPP: 22m 39s		61h 02m 24s
Morton Order	HSS: 4h 46m 12s	40m 17s	5h 26m 29s
	HPS: 2h 10m 45s		2h 51m 02s
	HSP: 4h 35m 21s		5h 15m 38s
	HPP: 1h 31m 20s		2h 11m 37s

Legend	*H...Heap Sort*	*S...Serial Sort*	*S...Serial Merge*
	R...Radix Sort	*P...Parallel Sort*	*P...Parallel Merge*

Table 4.1: Sorting and Build-Up times for the Vienna Stephansdom model.

CHAPTER 5

Conclusion

We presented a variety of sorting algorithms to sort point data used for rendering point clouds in order to speed up the model build-up process that is necessary to allow rendering in real time. In order to be able to sort data sets that are larger than main memory, an external merge sort algorithm is used. Sorting points by axis yields the best performance as long as the cache used during the model build-up is large enough, otherwise Morton order yields a better result. Radix sort performs better than heap sort, but can only be used when sorting by axis. With parallel processing, it is possible to increase the speed of the sorting process, especially when sorting by Morton order, as the corresponding comparison operation takes significantly longer than the comparison by axis coordinates.

Bibliography

[1] T. Chan. *Closest-point problems simplified on the RAM.* ACM-SIAM Symposium on Discrete Algorithms. 2002.

[2] M. Connor and P. Kumar. *Fast construction of k-nearest neighbour graphs for point clouds.* IEEE Transactions on Visualization and Computer Graphics. 2009.

[3] C. Dachsbacher, C. Vogelgsang and M. Stamminger. *Sequential point trees.* ACM Trans. on Graphics, 22(3):657-662, 2003.

[4] F. Duguet and G. Drettakis. *Flexible point-based rendering on mobile devices.* Computer Graphics and Applications, 24(4):57-63, July-Aug 2004.

[5] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic (IEEE Std 754^{tm}-2008).* 2008.

[6] E. Gobbetti and F. Marton. *Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models.* Computers & Graphics, 28(6):815–826, 2004.

[7] M. Herf. *Radix Tricks.* 2001. http://stereopsis.com/radix.html (Accessed on 20.02.2013).

[8] J. Kontkanen and E. Tabellion and R.S. Overbeck. *Coherent out-of-core point-based global illumination.* Proceedings of the Twenty-second Eurographics conference on Rendering, pages 1353–1360, 2011.

[9] M. Levoy and T. Whitted. *The use of points as display primitives.* Technical Report TR 85-022, 1985. The University of North Carolina at Chapel Hill, Department of Computer Science.

[10] S. Marek. *Normal Estimation of Very Large Point Clouds.* 2011.

[11] G.M. Morton. *A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing.* Technical Report, Ottawa, Canada: IBM Ltd. 1966.

[12] R. Pajarola, M. Sainz and R. Lario. *XSplat: External Memory Multiresolution Point Visualization.* IASTED Visualization, Imaging and Image Processing, 628–633, 2005.

[13] S. Rusinkiewicz and M. Levoy. *QSplat: A multiresolution point rendering system for large meshes.* In Proc. ACM SIGGRAPH 2000, pages 343-352, 2000.

[14] C. Scheiblauer. *Hardware-Accelerated Rendering of Unprocessed Point Clouds.* 2006.

[15] C. Scheiblauer, M. Wimmer. *Out-of-Core Selection and Editing of Huge Point Clouds.* Computers & Graphics, 35(2):342-351, April 2011.

[16] P. Terdiman. *Radix Sort Revisited.* 2000. http://codercorner.com/RadixSortRevisited.htm (Accessed on 20.02.2013).

[17] M. Wand, A. Berner, M. Bokeloh, A. Fleck, M. Hoffmann, P. Jenke, B. Maier, D. Staneker and A. Schilling. *Interactive editing of large point clouds.* In Proceedings of Symposium on Point-Based Graphics (PBG 07), 2007.