Hardware Accelerated Rendering of Unprocessed Point Clouds

Claus Scheiblauer

Institute of Computer Graphics and Algorithms
Vienna University of Technology
Vienna / Austria

Abstract

In this paper we present a method for rendering unprocessed point clouds using commodity hardware. A point cloud is a set of coordinates which are interpreted as points in space. No assumptions have to be made for the point clouds, and therefore the point clouds do not have to be preprocessed. The method includes view-frustum culling and a level-of-detail (LOD) algorithm which does not need any additional geometry to the original point cloud. It renders always faster than the simple usage of vertex buffer objects (VBOs). Data inside VBOs reside in graphics card memory and can therefore be accessed very fast by the GPU.

Keywords: real-time rendering, level-of-detail algorithms, point-based rendering

1 Introduction

Points as rendering primitives are fairly new and were first mentioned in 1985 by [4]. From then on it took more than 10 years that points were considered as an option as geometric primitives. The main reason is that many points are needed to represent models with point clouds, and the sizes of such point clouds can be hundreds of millions of points. Only recently the computers have become powerful enough to handle such large point clouds. This is a matter of memory consumption and rendering performance.

Looking at polygon-based models, the geometric details can be raised to a level where the sizes of the polygons sometimes even fall below the size of a pixel on screen. In such cases it would be a waste of resources to use a polygon mesh and perform the setup process needed for rendering.

Another example where points can be used for rendering is the output of a laser scanner. A scanner measures the distance of objects on its scanning hemisphere. These measurements can easily be converted into a point cloud. When the scanner position and the angle between successive samples is known, an approximation of the surface of the scanned object can be computed. It is an approximation in the sense, that no surface fitting algorithm is applied to the point cloud, rather the point sizes are adjusted to fill the spaces between the point samples (see section 5). Point clouds can be used for different tasks, and it is desirable to render them as fast as possible.

The rest of the paper is organized as follows. Section 2 gives an overview of current papers related to fast point-based rendering. Section 3 lists the algorithms that were implemented from papers. In section 4 our newly developed algorithms are introduced. Section 5 describes a fake surface rendering method. Section 6 gives an overview of the performance of the different algorithms. Finally section 7 presents the summary of this paper.

2 Related Work

The QSplat algorithm [5] uses a very compact data structure for storing and rendering point clouds. The input for the algorithm are either point samples from a laser scanner, or models consisting of polygons which are then sampled and represented as a point cloud in a preprocessing step. The layout of the data structure is a bounding sphere hierarchy. Every child bounding sphere is completely surrounded by its parent. The points within the hierarchy contain averaged informations, like colors and normals, from their children. So an intermediate node represents all informations from its children. These intermediate nodes are used during rendering for a LOD mechanism, where the recursion only steps down a level in the hierarchy if the projected size of the bounding sphere of the current intermediate node covers more of the screen then a certain threshold, else a splat with the attributes of the current intermediate node is drawn. The coordinates and attributes of the points are quantized, and therefore need only little space. But this is a trade-off, because the encoding of the positions require the algorithm to use the CPU for decoding the information during rendering.

The ρ-grids algorithm [2] is a means of rendering point clouds on mobile devices that do not have a FPU. The rendering pipeline of this algorithm is software based. ρ-grids are a generalization of an octree. An octree divides a cube into 2x2x2 equal-sized smaller cubes, whereas a ρ-grid divides a cube in ρxρxρ equal-sized smaller cubes. The memory on mobile devices is limited, and the ρ-grids provide a very memory-efficient data structure for storing point clouds. In a ρ-grid the positions of points are implicitly encoded as the filled cells of the ρ-grid. During rendering, the center of each cell is computed in a

clever way, which only needs additions and no multiplications. When a leaf node is reached, a point is drawn at the center of the leaf cell. A level-of-detail (LOD) algorithm is also included. LOD is used to approximate the model with fewer primitives when it is viewed from a distance. During build-up of the hierarchy the inner nodes receive a color that is the averaged color of their children cells. During rendering a screen-space bounding rectangle for each cell is calculated, and if the bounding rectangle is smaller then a pixel the traversal stops at the current node. When the current node is an inner node, the point with the color of the inner node is rendered to the screen. All other nodes further down the hierarchy don't have to be processed. The screen-space bounding rectangle can also be used for view-frustum culling.

The Sequential Point Trees (SPT) algorithm [1] uses an octree as hierarchical data structure, which is sequentialized so that it can be processed on the GPU. The points will be visualized as rectangular splats by using non-antialiased OpenGL points. When traversing a hierarchy, it is implicitly known if an ancestor node has been rendered, because then the dependent subtree will not be processed. If the data structure is sequentialized, it is not known if an ancestor has already been rendered, so the algorithm has to check for this case as well. The SPT algorithm calculates an r_{min} and an r_{max} for each point, which is the minimum and maximum distance of a point to the viewpoint for which the point will be rendered. These distances can be checked by a vertex program on the graphics card. If the SPT is sorted by r_{max}, then the CPU can cull those nodes whose r_{max} value is too small for the current viewpoint, so that they will not be rendered. It then sends all points from the first one in the r_{max} ordered list to the last one that will not be culled to the GPU. The SPT is sorted by r_{max} only once in a preprocessing step. All other computations can be done directly by the GPU. The SPT is stored as VBO. A disadvantage of the SPT algorithm is that within an SPT, no view-frustum culling can be applied. Another disadvantage is that most of the time too many points are sent to the GPU, from 10 to 40 percent. This can be avoided, as described in section 4.3.

The Layered Point Clouds (LPC) algorithm [3] also uses a very fast rendering algorithm. The input for the LPC is a set of evenly sampled points. Then a hierarchy is built up such that the points at each level are also evenly distributed. The sum of all levels of this point cloud's hierarchy, starting at the root node, represents the whole model at a certain level without producing new nodes. Each level refines the representation from the upper levels. The hierarchy is divided into an index tree and a point cloud repository.

The split of the data structure into an index tree and a point cloud repository makes it possible to influence many points with one decision, which significantly reduces the time for traversal. The point clouds are then rendered as VBOs. For large models, not all point clouds can be stored on the graphics card. Therefore the algorithm manages a least recently used cache to minimize the lag when swapping VBOs in and out of graphics card memory.

The algorithms in this paper are limited to models that fit in the graphics card memory. Out-of-core rendering will likely be a task for future enhancements.

3 Implemented Algorithms

In the search for a fast point-rendering algorithm we were looking for algorithms that optimize the rendering speed, that is the frames per second (FPS), and the throughput, that is million vertices per second (VPS). VPSs can be used to check if an algorithm uses the GPU efficiently. On new graphics boards performance counters can be used to easily determine the VPS.

Many algorithms were tested with different strengths and weaknesses. From a software-based rendering pipeline [2] to a completely hardware-accelerated approach [1]. The advantage of a software-based rendering pipeline is that the points can be stored in a compact representation which can be decoded during run-time. The disadvantage is that a software-based rendering pipeline is CPU-bound, and that the processing power of the GPU is neglected. The advantage of a completely hardware-accelerated approach is that the CPU is relieved from processing needs, but this can also be a disadvantage in some situations. Today it is not possible to implement view-frustum culling for a hierarchy that completely resides in the graphics card's memory. For a fast point-rendering algorithm, the CPU is still needed to do some preprocessing before rendering the points to screen.

3.1 Vertex Buffer Objects

The simplest way when trying to render a point cloud as fast as possible is to use VBOs. Vertex buffer objects are OpenGL vertex arrays stored in graphics card memory. The main drawback of using simple VBOs is that they will always draw the complete model, independent of the current viewpoint. If the model is viewed from some distance, so that some points will be projected to the same pixel and therefore the pixel is overdrawn multiple times, the rendering will become even slower. This is why a level-of-detail algorithm is desirable. Also view-frustum culling is not possible when there is no supporting data structure.

3.2 ρ-grids

We next evaluated the ρ-grids [2] algorithm. Although the rendering pipeline is software based, the level-of-detail algorithm is very efficient for viewpoint positions in the distance. To build up a ρ-grid, the points from the original point cloud are sorted into the grid. When only one point is left in a cell, or a user defined recursion depth is reached, the cell becomes a leaf node. This way the point cloud

is resampled, and the points are represented as the cell-centers of a hierarchical grid.

We modified the rendering pipeline for our implementation. In contrast to the original paper, during rendering the cell-centers are calculated in object-space coordinates as well as in clip-space coordinates. The clip-space coordinates are used for view-frustum culling in clip-space, and the object-space coordinates are used for the point positions, which are then in turn used for OpenGL `glVertex()` calls. The points are visualized as non-anti-aliased OpenGL points. To check if a cell appears smaller than a pixel on screen, we project the diameter of the bounding sphere of a cell to the viewport. If the diameter is smaller then a pixel, the cell's point is drawn and recursion stops. Using the diameter is a conservative approximation, as the diameter is larger then the side of a cell for most viewing positions. But it is never smaller.

Tests have shown that this method is most effective if the points of the model are clustered in a region in space, so that the ρ-grid is densely populated. If the model consists of points that are spread over some region, the ρ-grid becomes sparsely populated. This leads to an increase in the number of inner nodes, which can become the dominating part of the hierarchy. The memory requirements are potentially low, because no point coordinates are stored. But this could be a disadvantage, because during build up the model will always be resampled. The center of a cell only approximates the original coordinates of a point.

3.3 Sequential Point Trees

We also implemented the SPT algorithm [1], but with a simpler error measurement as the one described in the original paper. In [1] the error is made up of a perpendicular error, which tries to alleviate errors along the silhouette, and a tangential error, that measures how well a parent cell covers the children cells. For these error measures a normal vector for each point is required. For our algorithm we only check if the bounding sphere diameter of an inner node appears smaller than a pixel from the current viewpoint or not. If it appears smaller than a pixel, the recursion can stop at the inner node. For rendering an SPT, first an octree is built up in memory. Then the octree is sequentialized and saved as an array.

An SPT is an ordered sequence of point coordinates, colors, and octree recursion level indices, which are all needed for rendering. The ordering is done for increasing recursion level. So all points that would first be rendered at level i are tightly packed in this array, and after that all points that would first be rendered at level $i+1$ follow. The hierarchy and the sequentialized version of the octree can be seen in figure 1. For maximum performance this array is also saved in a VBO.

The rendering of an SPT is as follows. The SPT algorithm checks down to which level the hierarchy would have to be traversed, so that a cell would be smaller than a pixel on screen. This is then the level down to which

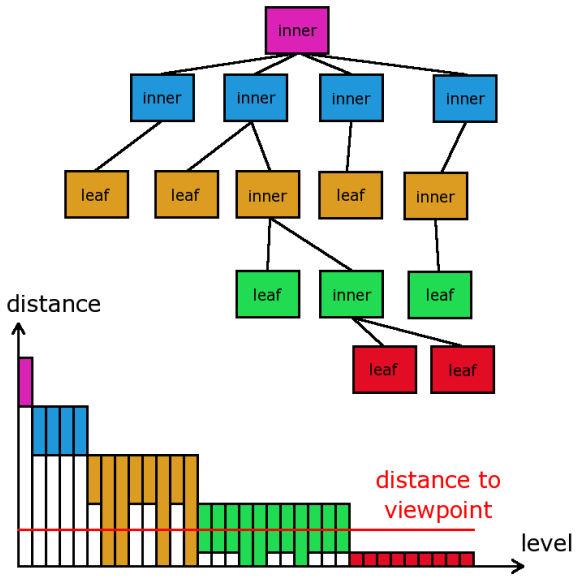

Figure 1: The upper figure shows the hierarchy as a tree, the lower figure is the sequentialized version. When leaf nodes at a recursion level exist, they will be rendered from that distance until the viewpoint is very close to the model.

points in the SPT are selected. This means that for viewpoint positions that are in some distance of the model, not all points in the VBO will be rendered. On the GPU a vertex program is needed which has to decide if a point should be rendered. The decision is done per point, and it is checked if it lies within its allowed range. The range can be derived from the recursion level at which the point lies within the octree (see figure 2). For this the vertex program has stored the distances for all recursion levels. The minimum and maximum recursion level for a point are sent to the vertex program. There is no overlapping distance between inner points from different recursion levels. This is due to the simpler error measurement. The indices for the recursion levels need memory and processing time, and with some reordering of the data they can even be avoided (see 4.3).

In figure 1 the red line marks some arbitrary distance of the viewpoint to the cell that holds the SPT. Within the orange and green recursion depths the inner points and leaf points are mixed. All points from the left side of the sequentialized hierarchy on to the green recursion level are sent to the GPU. The magenta, blue, and orange inner nodes will be culled by the vertex program. Figure 2 shows the areas in which the inner nodes from the different recursion levels will be used. The leaf nodes will be rendered as soon as the viewpoint crosses the border to their recursion level area. When the viewpoint continues to the next recursion level area, the leaf nodes of the previous recursion level will still be drawn. The red circle corresponds to the red line in figures 1, 4, and 5.

One deficiency of the SPT algorithm is that it cannot perform view-frustum culling. Its main advantage is

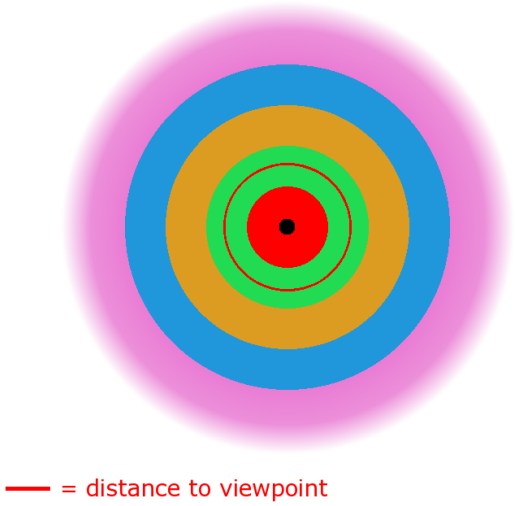

Figure 2: The black spot is the model as seen from above. The colored areas around the model denote the recursion level down to which the traversal goes, when the viewpoint is in some distance to the model.

the speedup in rendering performance when looking at a model from some distance. The speedup for viewpoint positions in the distance is big. But for viewpoint positions that are within the model, like in a walkthrough, the overhead of the additionally rendered intermediate points cannot be neglected.

4 Developed Algorithms

The implemented point rendering algorithms in section 3 all have their limitations. So we tried to develop an algorithm by combining a hierarchy that is traversed by the CPU to enable view-frustum culling, and to store the points in graphics card memory, so they can be rendered as fast as possible.

4.1 Vertex Buffer Objects revisited

For the next algorithm we inserted VBOs with points in an octree hierarchy. For the algorithm we use the original points of a point cloud. From the point cloud an octree hierarchy is built up, where each leaf node holds only one point, or until a user defined depth is reached. This octree is saved to harddisk. For rendering, the octree is again built up in memory, but all points that are in leaf nodes are put in VBOs. When during build-up of the rendering hierarchy the number of leaf nodes below an inner node is within a user defined threshold, then all points that are in the leaf nodes are copied to the VBO. The inner nodes for this part of the hierarchy are ignored. In figure 3 a subtree of an octree hierarchy is shown. If the maximum size of a VBO is 20 points, then branches a) to d) are in different VBOs. The nodes at recursion depth 1 are the VBO nodes of the octree hierarchy, as they hold the informations for the VBOs. At recursion depth 0 the number of leaf points is greater than 20. At recursion depth 1 the number of leaf points is smaller than 20 for all nodes.

The traversal during rendering now is as follows. The hierarchy is the same as in the ρ-grid algorithm (see 3.2). When a leaf node is reached, and it lies within the view-frustum, the VBO is rendered with all points that it contains. The main disadvantage of this algorithm is that the leaf nodes are rendered too early. The number of points that are allowed in one VBO can not be set arbitrarily low, because rendering a VBO includes an overhead on the CPU for setup. The number of the VBOs has to be limited. Therefore the size of the octree cell that a VBO represents is rather big and will be larger than a pixel on screen, even if the viewpoint is still far away. So although the algorithm does contain a level-of-detail part, it is not efficient. Because of view-frustum culling, this algorithm is faster in some situations than the simple usage of VBOs (see 6).

4.2 Sequential Point Trees revisited

So next we tried SPTs instead of simple VBOs in the hierarchy to be able to use view-frustum culling combined with SPTs. Even a small SPT with only a few levels of the octree hierarchy has an performance improvement over a simple VBO. The hierarchy is built up and saved on the harddisk like in the previous section 4.1. When building up the hierarchy in memory for rendering, a whole subtree of the octree on disc is copied to a SPT, including the inner nodes. The user can define the maximum number of points that are allowed within an SPT. Compared to figure 3, all nodes of the shaded branches are used for the four

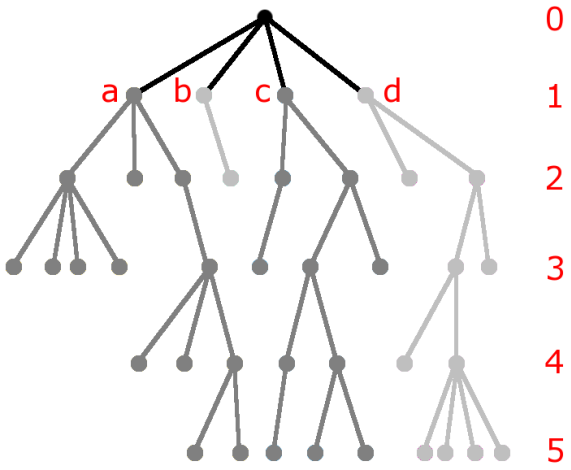

Figure 3: A subtree of an octree. The four shaded branches of the subtree (a-d) are stored in four different VBOs. On the right side the recursion depth of the subtree is noted.

different SPTs.

The traversal during rendering is the same as in section 4.1, but when a node containing an SPT is reached, the SPT algorithm is performed. Therefore not all points have to be rendered when the viewpoint is in some distance to the model.

We observed improved performance compared to the algorithm of section 3.3. The view-frustum culling allows for fast rendering when the viewpoint lies within the model. In some situations, when the viewpoint is in the middle of the model, the algorithm with VBOs inserted into the hierarchy is faster. This is possible because the VBOs contain less points as when the model is rendered with SPTs in the hierarchy.

4.3 Enhanced Sequential Point Trees

The VBOs included in a hierarchy are fast for viewpoints that lie within a model, as it is used in a walkthrough, and SPTs included in a hierarchy are fast for viewpoints that lie in a distance to a model, as it is useful when approaching a model from the outside. Our next algorithm combines the advantages of these two methods. An effect of the combination is that the memory requirements can be reduced.

The original SPTs mix up the inner nodes and leaf nodes of the built up octree in one VBO (see figure 1). The difference between inner nodes and leaf nodes is that the inner nodes are only rendered for a part of the distance when approaching a model (see figure 2). When inner nodes and leaf nodes are combined in one VBO, a vertex program is needed to decide if the inner nodes should be rendered or not. Leaf nodes will never be culled by the vertex program. To speed up processing during rendering, inner

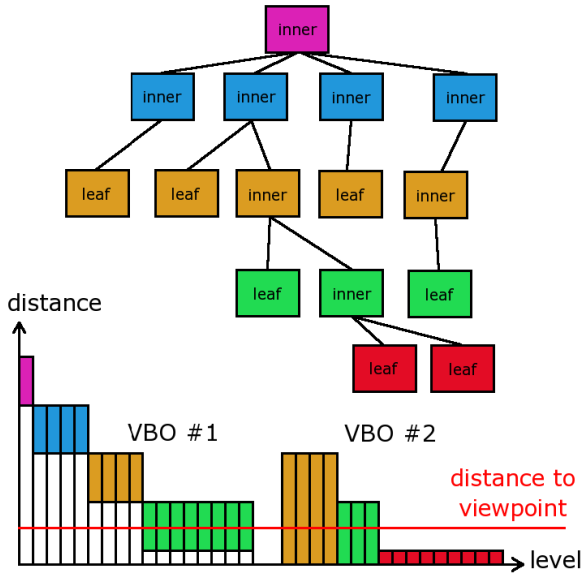

Figure 4: For the enhanced SPTs the inner nodes and the leaf nodes are separated into two different VBOs.

nodes and leaf nodes of one SPT can be copied to two different VBOs. The points in both VBOs are still ordered by increasing recursion level. With this rearrangement it is now possible to make the decision which inner nodes should be rendered completely on the CPU.

Since the error measurement in the original SPT also pays attention to the curvature of the surface, the rendering distances of the inner nodes are not equal for cells of the hierarchy. With the simpler error measurement, the inner nodes of one level are rendered exactly for the same distance. There is no overlapping distance necessary when the viewpoint moves between the distances where different recursion depths of inner nodes are used. In figure 2 the recursion level is changed exactly on the borders, if the viewpoint moves towards or from the model.

The rendering is a little bit more complicated then before. The inner nodes of the hierarchy are still calculated like in the ρ-grid algorithm (see 3.2). When a leaf node is reached, and it lies within the view-frustum, the decision has to be made whether the VBO with the inner nodes is needed at all. If it is needed, the recursion level is calculated down to which the hierarchy would have been traversed, and the inner nodes for only this recursion level are selected. This situation is shown in figure 4. The distance of the viewpoint to the model is such that the level with the green inner nodes is selected from the VBO #1. From the VBO #2 all leaf points up to the green level are rendered. If no inner points are needed, only points of the VBO with the leaf nodes are rendered. Due to the rearrangement, no vertex program is needed, which also means that no minimum and maximum indices for each point is required. The selection of the inner nodes is done on the CPU. The number of the rendered points is always lower or equal to the number of the original points.

The presented enhancements cut the memory requirements of the SPT by 20 percent, which means that larger models can be visualized (see section 6).

4.4 Memory Optimized SPTs

The enhanced SPTs still have one deficiency with respect to the memory requirements, and that are the additionally created inner nodes. The effect of them is that the model will look from the distance as if it was filtered by a low pass filter. This is similar to mipmapping without interpolation between mipmap levels. The inner nodes are also used for the level-of-detail algorithm. An inner node is rendered when the cell is smaller than a pixel on screen.

The idea for the memory optimized sequential point trees is to use some of the original points as the points for the inner nodes. For this it is possible to use any point of an octree cell, because if the cell appears smaller than a pixel on the screen, then any point that lies within the octree cell will be rendered at the same position. For the presented algorithm an arbitrary choice is made, and the point that is closest to the center of the cell is chosen as the point for an inner cell. The selection is done during the initial

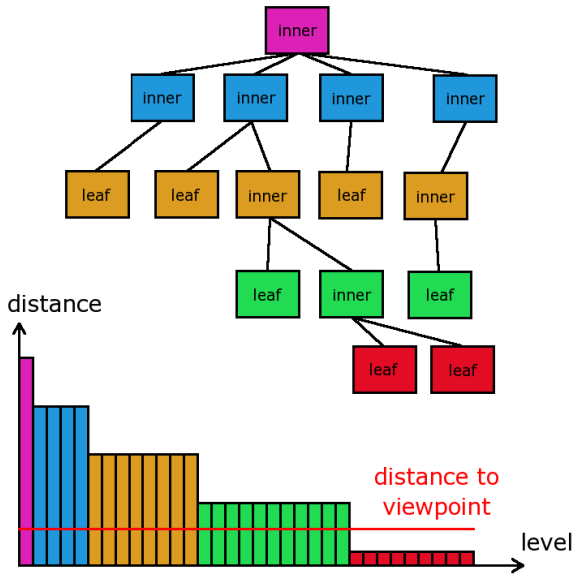

Figure 5: Memory optimized SPTs use for the inner nodes and for the leaf nodes points from the original model. Once the distance for a recursion level is passed, the points from the inner nodes and from the leaf nodes will always be rendered.

build up of the octree from the unorganized point cloud. For the initial build up one point after the other is inserted into the octree. If a point is closer to the center of a cell than the point that is currently used as an inner point, then the points are swapped, and the former inner point is now used as the point that has to be inserted to the octree. It is sufficient to continue at the cell where the points were swapped, because the former inner point will certainly be a part of that cell.

The rendering hierarchy consists of an octree with VBOs as leaf nodes, which contain a sequentialized version of the subtree they represent. Because the original points are used for the inner nodes, the hierarchy uses 50 percent less memory than the original SPT algorithm, and 35 percent less than the enhanced SPTs. This can be an advantage for very large models (see 6).

The rendering is simplified compared to the algorithms before. Because there are no additionally created points, the inner points of the outer octree runtime structure can also be collected in a VBO and are always drawn to screen. This is reasonable, because the total number of inner nodes is only a few thousand, even for very large models (see 6). Therefore it is faster to always draw a VBO then to perform view-frustum culling for each point and render it between a `glBegin() ... glEnd()` clause. The octree hierarchy is traversed nevertheless, and if a cell appears smaller then a pixel on screen, the recursion simply stops. When a leaf node is reached, and it lies within the view-frustum, the SPT algorithm is performed, and here again no vertex program is needed. The algorithm is the same as rendering only the ordered leaf points VBO for the enhanced SPTs (see figure 5). When looking at figure 3, then all nodes would contain original points. The node at recursion depth 0 is part of the unordered VBO that will always be rendered if the model is within the view-frustum.

The memory requirements are comparable to the algorithm with VBOs inserted into the octree. But in contrast to this former mentioned algorithm, the level-of-detail part works efficiently.

5 Dynamic Point Size

When a range scan is used as the original point cloud, then the surfaces of objects in the scan can be approximated. It is not a high-quality approximation, but it performs quite fast. From a range scan the angle between two consecutive samples is known, and a point with a size that is large enough to close the space between two samples can be rendered. The point size is calculated for each SPT in the hierarchy, using the sample that is farthest away from the scanning position within the SPT. OpenGL needs the screen-space point size, and this one is derived from the object-space point size during rendering. The calculation during rendering is not optimal for the performance, but adaptive to viewport resizing.

6 Results

The performance tests are a bit complicated, because the algorithms show their characteristics only if different situations are examined. As a typical model a scan from the Stephansdom project is chosen for the performance tests. In figure 6 the outer border of the model, as seen from above, is visualized as black circle. Within this circle the scanner has taken point samples. The different positions where the performance measurements were

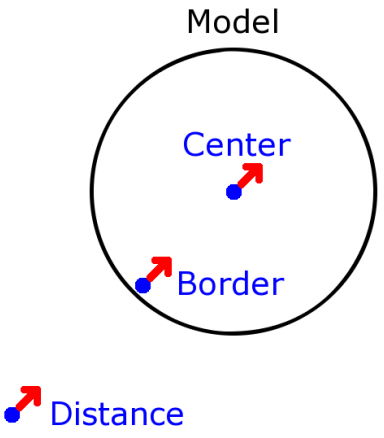

Figure 6: The different positions relative to the model, where the performance measurements were taken.

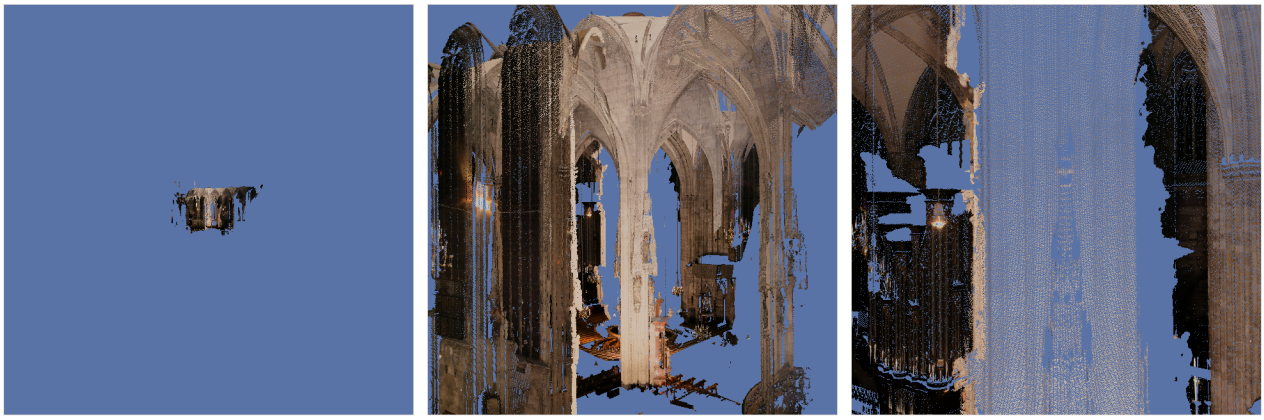

Figure 7: The model as seen from the different measurement positions. On the left side a) shows the model from a distance. In the middle b) shows the model from just within the border of the model. On the right side c) shows the model as seen from the center. The viewing direction for all positions is the same.

taken are marked as blue spots. The red arrows symbolize the viewing direction. The viewing direction is constant throughout the measurement process. The three different viewing positions as seen in the OpenGL viewport are depicted in figure 7. The viewport has a size of 640x640 pixels. It shows the model rendered with the memory optimized SPTs (see 4.4). The visual quality of all algorithms is nearly the same, only when the model is in great distance to the viewpoint then noticeable differences appear. For the performance measurements we used a computer with a 3,2 GHz Intel Pentium 4 processor and a NVIDIA 6800 GTO graphics board. The graphics board incorporates 5 vertex shader units. This is just enough so that the vertex program for the SPTs is not the bottleneck in the rendering pipeline.

For maximum throughput it was necessary to optimize the API usage of OpenGL. The vertex program uses the ARBVP1 profile, which is the simplest profile, but the vertex program compiles to less instructions as when using a more complex profile and executes faster. Another issue was the data layout for the VBOs. A VBO is rendered most efficiently when the number of bytes for the attributes of one vertex is 16 or 32. This means when 3 attributes like vertex coordinates, color and recursion level indices are needed, then vertex coordinates and recursion level indices are stored as one intertwined array, which needs 12 bytes for the coordinates and 4 bytes for two short integers for the indices. The color with 4 bytes is stored as an extra array. The last issue was the binding overhead for a VBO on the CPU. The solution to this is to create one large VBO and store the data arrays at different offsets. The offsets are aligned to 32 byte, because if they are not aligned a performance penalty occurs when reading the data for rendering.

Algorithm	Distance	Border	Center
OneVBO	15	17	18
OneSPT	239	12	12
VBOsR	15	18	228
SPTsR	286	21	160
ESPTs	378	31	240
MOSPTs	300	28	228

Table 1: Frames per second for the different algorithms at different positions.

Algorithm	Distance	Border	Center
OneVBO	6609K	6609K	6609K
OneSPT	456K	10021K	10021K
VBOsR	6522K	6236K	480K
SPTsR	363K	5357K	696K
ESPTs	268K	3683K	454K
MOSPTs	356K	4065K	481K

Table 2: Numbers of points for the different algorithms at different positions. K stands for thousand.

We compare the algorithms of sections 3.1 (“OneVBO”), 3.3 (“OneSPT”), 4.1 (“VBOsR”), 4.2 (“SPTsR”), 4.3 (“ESPTs”), and 4.4 (“MOSPTs”). The OneVBO algorithm can be seen as the benchmark which we try to beat with our developed algorithms. The VBO size is limited to 10000 points for all algorithms except for the OneVBO and the OneSPT, which use one VBO for all points. The point size for all measurements is 1.

Table 1 shows that the OneSPT performs very good for viewpoints in the distance, but very bad for viewpoints within the model, because then all points that are contained in the SPT have to be processed (see table 2). This behavior can also be seen with the SPTsR algorithm, which needs more points and renders slower than the VBOsR for viewpoints in the center of the model. The MOSPTs and ESPTs algorithms are the fastest for any

Algorithm	Distance	Border	Center
MOSPTs + dps enabled	167	27	224
MOSPTs + dps filling	167	27	214

Table 3: Frames per second when dynamic point size is enabled without producing larger points (upper row), and when dynamic point size is used to fill the spaces between neighboring points (lower row).

viewpoint. The MOSPTs algorithm seems to be a less optimal method than using artificially created points for the inner nodes as the ESPTs algorithm does.

The VPS for all algorithms is nearly the same, and close to the theoretical maximum of the graphics board. According to NVIDIA the graphics board has a theoretical maximum of 116 million VPS, and we observed VPS rates from 113 to 114 for all algorithms. The VPS is measured at the border position, because there the most points are rendered, which hopefully minimizes the impact of the hierarchy traversal.

Table 3 shows the difference in the FPS when using dynamic point size (DPS). In the center the difference between enabled DPS, which renders points with size 1, and DPS that fills the spaces between the points is 5 percent, which is due to the higher fill rate.

Figure 8 gives a comparison of the total number of points that reside in the hierarchy of the different algorithms, and the memory requirements for the hierarchy on the graphics board. The algorithms are ordered by the minimum FPS that an algorithm has achieved in the comparison, so this is some kind of classification. Only the MOSPTs and the OneVBO do not need any more points, than in the original model are. For the VBOsR the inner nodes of the octree hierarchy require additionally created points. There are 500 of them.

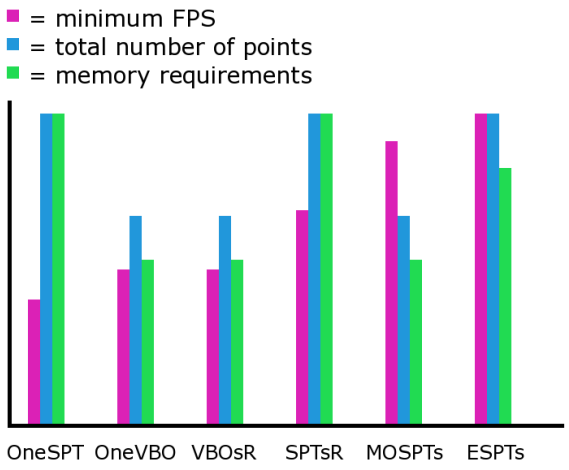

Figure 8: Algorithms ordered by minimum FPS. Maximum minimal FPS equals 31, maximum number of points equals 10021473, maximum memory requirements equals 202 million bytes. The bottom line is at zero.

At last a very large model, which does not fit into graphics card memory, was tested. It contains about 28M points. Only some 2000 points are not part of an ordered VBO and reside within the octree hierarchy. The model uses point samples from 10 different scanning positions. Points far away from the model were cut by hand, and the density of the points was reduced, so the point cloud for this model is preprocessed. For the test only the memory optimized SPTs were used. With this algorithm it is possible to maintain a throughput of 80M VPS, but this also indicates, that points needed to be fetched from main memory over the PCI-E bus. At some positions about 15M points are visible, which reduces the FPS to 5.

7 Conclusions

In this paper we have presented a method for rendering unprocessed point clouds, using only the points of the original model. The method is fast for viewpoints from the distance and for viewpoints within the model. We compared the algorithm to different implementations of the SPT algorithm. The new method is not the fastest, but memory efficient. Future enhancements to the algorithm could involve an out-of-core part, because in this implementation the algorithm is not well suited for very large models.

References

[1] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. In Jessica Hodgins and John C. Hart, editors, *Proceedings of ACM SIGGRAPH 2003*, volume 22(3) of *ACM Transactions on Graphics*, pages 657–662. ACM Press, 2003.

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

[3] Enrico Gobbetti and Fabio Marton. Layered point clouds. In Marc Alexa, Markus Gross, Hanspeter Pfister, and Szymon Rusinkiewicz, editors, *Eurographics Symposium on Point Based Graphics*, pages 113–120, 227, Aire-la-Ville, Switzerland, June 2004. Eurographics Association. Conference held in Zurich, Switzerland, June 2–5, 2004.

[4] Mark Levoy and Turner Whitted. The use of points as a display primitive. *Technical Report TR 85-022*, 1985. The University of North Carolina at Chapel Hill, Department of Computer Science.

[5] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 343–352. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.