Eurographics Symposium on Point-Based Graphics (2006)
M. Botsch, B. Chen (Editors)

Instant Points: Fast Rendering of Unprocessed Point Clouds

Michael Wimmer[†] and Claus Scheiblauer

Vienna University of Technology, Austria

Abstract

We present an algorithm to display enormous unprocessed point clouds at interactive rates without requiring long postprocessing. The novelty here is that we do not make any assumptions about sampling density or availability of normal vectors for the points. This is very important because such information is available only after lengthy postprocessing of scanned datasets, whereas users want to interact with the dataset immediately. Instant Points is an out-of-core algorithm that makes use of nested octrees and an enhanced version of sequential point trees.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Display algorithms, Viewing algorithms

1. Introduction

Point-based rendering has gained a lot of popularity due to the availability of 3D range scanners. Recent long-range laser scanners capture data up to 1000m, giving wildly varying sampling densities (Fig. 1). Practically all current point rendering approaches rely on assumptions about sampling densities, or the availability of normal vectors for each point. In fact, most papers present models that have been gained by point sampling a geometric mesh.

Unfortunately, post-processing a scanned point cloud to obtain a mesh can take several person months. This is because especially data acquired in the outdoors is not amenable to automatic postprocessing methods. Such data is characterized by holes in almost any surface, varying sampling densities, and a mixture of surface and non-surface structures (such as leaves), which is a problem for meshing approaches. There are several efficient methods to estimate the normal vectors required for point rendering techniques directly from the point cloud [TKDS05]. However, normal estimation assumes that the points represent sufficiently dense samples of an underlying surface, which cannot be assumed for long range scans, where the data is simply too sparse in many regions. Thus, lengthy manual postprocessing is inevitable.

On the other hand, there is a real need for users to explore and interact with the scanned data instantly. This is a factor of both costs and opportunity: paying several months for a qualified engineer to postprocess a scanned model can be prohibitive for most potential users (e.g., archaeologists, architects et.) of 3D scanners. Furthermore, at the time the post-processed model is available, the original research question might have already been solved in another way, rendering the model unnecessary.

This is where *Instant Points* come in: We present the first point-rendering algorithm that does not require postprocess-

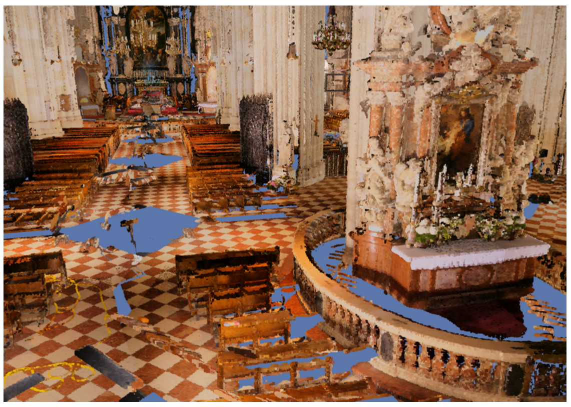

Figure 1: *Example screenshot of an unprocessed point cloud consisting of 262M points.*

† wimmer@cg.tuwien.ac.at

ing of point clouds and which at the same times renders enormous amounts of *unprocessed points* at interactive rates with negligible preprocessing. With unprocessed point clouds we mean those which have not been *interpreted* in any way (meshing, normal estimation etc.), i.e., each point is defined only by a 3D position and—if available from the scanning process—an RGB-color. The point cloud is converted into an efficient out-of-core structure which is based on a combination of nested octrees and memory optimized sequential point trees, optimally exploiting current graphics hardware.

Unprocessed point clouds evidently cannot match postprocessed models in image quality. The main contribution of this paper is to show how to trade this reduced image quality against significantly increased visualization speed and improved memory requirements. There is a definite need for unprocessed point rendering, starting from quick onsite visualizations during the scan campaign (which can help in scan planning, scan verification etc.); visualization systems where the general "feel" of a location is more important than the exactness of every minute detail; experimental systems for archaeologists, architects and regional planners, for which such unprocessed point rendering would make range scanning viable at all; visualization systems where access to the original point cloud is needed; and many others.

2. Previous Work

Rendering point clouds has recently become a popular topic. However, the huge majority of research has concentrated on rendering datasets that have been postprocessed, if not even sampled from an original geometric model. These rendering approaches build on the pioneering Surfel approach [PZvBG00], which basically defines a point as a small ellipsoidal disk with normal vector. The trend is towards ever higher-quality rendering methods for Surfels, as presented by Botsch et al. [BHZK05], who also give a very good overview of previous high-quality point splatting methods.

However, high-quality input models are not often available, and methods dealing with unprocessed point clouds are rare. Xu and Chen were the first to realize the extreme difficulty of displaying models acquired from long range outdoor scanners. Instead of trying to postprocess the data, they propose to use non-photorealistic rendering techniques in order to change the viewers' expectations on the realism of the viewed model [XC04]. While their system produces stunning results in many cases, it is limited to smaller datasets due to its computational complexity.

Sequential point trees (SPT) by Dachsacher et al. [DVS03] is one of the fasted algorithm for rendering small to medium point clouds because it makes good use of graphics hardware. They realize that maximum throughput can only be achieved if large batches of primitives are stored in graphics hardware buffer objects. For unprocessed point clouds, however, the representation incurs significant memory and performance overhead as will be shown in Section 4.

The first system that was able to render large point sampled models that do not fit in main memory was QSplat [RL00], which builds a hierarchy that allows per-node level-of-detail (LOD) selection. However, the selection has to be done on the CPU. A similar approach is taken by Duguet and Drettakis [DD04], who also use an LOD representation of a model that is processed per node. They target the special hardware of PDAs, which do not have dedicated graphics processors. The layered point cloud (LPC) [GM04] system uses block LODs for point sampled models and achieves rendering rates that are an order of magnitude higher compared to QSplat by making efficient use of graphics hardware. The system assumes a uniform sampling density of the input data. Our nested octrees are similar in spirit to layered point clouds, but our algorithm works on arbitrary data which is not necessarily sampled uniformly. XSplat [PSL05] is another system for out-of-core rendering of huge point clouds. Similar to our Instant Points system, XSplat is based on SPTs and uses a two-level hierarchy. The main difference is that XSplat is aimed at rendering high-quality models, whereas Instant Points offers significant optimizations for unprocessed point clouds. Out-of-core methods have also been extensively studied for triangle rendering, where two-level hierarchies are used as well [YSGM05], but in addition, connectivity has to be taken into account.

Some systems approximate (parts of) the point cloud with textured [WGK05] or normal-mapped [BDS05] polygons without extracting the topology. While these approaches can provide extremely fast frame rates for large point clouds, we opted for a system where the original scanned points can be shown at the finest level. A promising alternative to out-of-core rendering is to compress the point cloud so that it fits into graphics card memory and can be decoded directly on the GPU [KSW05].

3. The Instant Points Rendering System

Unprocessed point clouds do not contain any connectivity or density information, making it difficult to devise a suitable rendering representation. The main idea of the Instant Points system is to circumvent this fact by interpreting point samples through the effect they have on rendering in a frame buffer. This means that if more than one point projects to one pixel, only one actual representative point is needed to fill that pixel. More concretely, we deal with the problem of viewing a dense point cloud, which can be solved through subsampling the point cloud. Without additional information, we can not solve the interpolation problem that arises when point clouds are viewed from too near a distance. In this case, we give the user a choice of using a fixed world-space extent for point samples, or a multiple of the sample distance which is derived from the depth of the hierarchy.

The Instant Points system consists of two main elements:

- Memory optimized sequential point trees (MOSPT), a version of SPTs improved for unprocessed point clouds.
- Nested octrees, a structure that allows out-of-core rendering, and contains MOSPTs as elements.

MOSPTs can be used alone for smaller point clouds to remove the 125% of memory overhead caused on average by SPTs for unprocessed point clouds, and increase the rendering speed by not having to render the 50% of additional interior nodes in the hierarchy, and to simplify the rendering process. However, we will mainly use them as parts of the nested octrees described later.

4. Memory Optimized Sequential Point Trees

4.1. Sequential Point Trees Revisited

Sequential Point Trees (SPTs) [DVS03] are a hierarchical point representation that allows rendering through a *sequential* processing by the GPU, while the CPU is available for other tasks. Each node of the hierarchy is associated with an error e. The recursive traversal checks whether the projected error $e/r < \varepsilon$, where r is the view distance. This can be simplified by storing a minimum distance $r_{min} = e/\varepsilon$ with the node, so that the test becomes $r > r_{min}$. In addition, each node also stores a distance r_{max}, in the simplest case the r_{min} of the node's ancestor. This allows a non-recursive test for each node by checking whether $r \in [r_{min}, r_{max}]$. This test can easily be carried out by a vertex program on the GPU. Additionally, by sorting the vertex list by r_{max} and calculating a lower bound $min(r)$ via the bounding sphere of the object, the GPU need only process a prefix of the vertex list with $r_{max} < min(r)$. Processing only this prefix is the main reason why SPTs are so efficient.

The error e in the original SPT algorithm assumes that the points are actually splats (or surfels) with a splat size d and a normal vector n. Inner nodes have splat sizes that encompass the child nodes. This allows levels of detail where larger splats approximate flat surface areas, and smaller splats are used in curved areas.

4.2. Screen Splat Error Metric

A point in an unprocessed point cloud does not represent a surfel (i.e., a splat with normal and radius), but a point sample that is rasterized by graphics hardware as a screen-space splat. For a given point hierarchy, a node should be rendered when further recursive traversal would not change the points that are rasterized. This is the case if the projected size of the node is smaller than a pixel.

In order to achieve this semantics for SPTs, we define the error e of a node as the diameter d of the node geometry. Therefore, $r_{min} = d/\varepsilon$, where ε needs to be adjusted depending on the camera parameters and the desired splat size. This allows unprocessed point clouds to be rendered using SPTs.

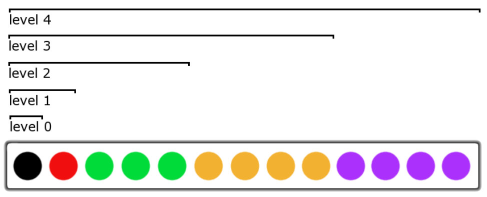

Figure 2: *Linearized MOSPT hierarchy. Higher-level nodes form part of lower levels of the hierarchy.*

4.3. SPT overhead

Each hierarchical data structure has a certain overhead depending on the average branching factor α. The *memory overhead M*, i.e., the number of interior nodes relative to the leaf nodes, of a hierarchical structure can be calculated as [DD04]:

$$M \sim \frac{1}{\alpha - 1}.$$

For an octree storing 2D surface data, $\alpha = 4$, resulting in an overhead of 33%. However, our experiments have shown that data coming from range scanners have lower branching factors. For several different datasets, the observed branching factor was $\alpha = 3$, resulting in an overhead of $M = 50\%$.

Note that this causes not only an increase in the memory required to store an SPT in graphics hardware, but it also directly reduces rendering performance, since the interior nodes are additional points that have to be processed by the vertex processor. Especially for viewpoints near the model, where $r_{max} > min(r)$ for all nodes and the whole vertex list has to be processed by the GPU, rendering time is increased by $M = 50\%$ on average in comparison to rendering the original points only.

Furthermore, nodes need to store r_{min} and r_{max}, taking 4 bytes (1 float) each. Assuming that for scanned datasets, only a point position (12 bytes) and a color (4 bytes) is stored normally, this would result in a *structural overhead* $S = 50\%$. In total, for unprocessed point clouds, the combined overhead M and S caused by SPTs is 125% on average, depending on the branching factor of the specific model.

4.4. Memory optimization

In order to overcome the significant memory and rendering overhead caused by SPTs for unprocessed point clouds, we make use of the following observation, which follows directly from the definition of the screen splat error metric:

Any child node of an interior node that is selected *for rendering will lead to the same pixel on screen being rasterized as the interior node.*

Therefore, instead of creating a new point to represent an

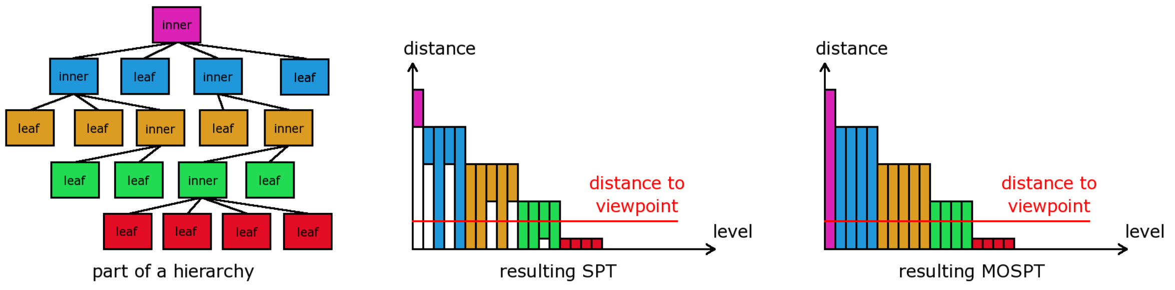

Figure 3: *A hierarchy (left) created for an SPT (middle) consists of original and additionally created points. The SPT contains inner and leaf nodes mixed in one level of the hierarchy. A hierarchy for an MOSPT (right) consists only of original points.*

interior node, we use an *existing child node* as a representative point for the ancestor. This is similar in spirit to vertex clustering algorithms [LE97, RB93], where all vertices in a hierarchy node are replaced by one representative vertex taken from the input vertices. For an uncolored unprocessed point cloud, the choice of representative is completely arbitrary. However, most interesting point clouds have color information. In this case, we select the color that has the smallest color distance to the average color of the child nodes.

The resulting hierarchy can be stored in an extremely efficient way as an SPT. The nodes are still sorted by r_{max}, but instead of storing all nodes for each level in the vertex list, we omit for each level those nodes that were chosen as representative point in the previous (upper) level, and are therefore already stored in the vertex list. This new, *memory optimized SPT* (MOSPT), does not require more memory than the original point cloud. In fact, the hierarchical SPT structure is achieved by cleverly reordering the original point cloud (see Figure 2).

4.5. Rendering MOSPTs

The screen splat error metric leads to several simplifications in the rendering of MOSPTs in comparison to SPTs.

- The screen splat error metric d is constant for each level of the MOSPT hierarchy, since all nodes at the same hierarchy level have the same diameter. Therefore, a cut of the hierarchy with $r = const$ gives exactly one level of the hierarchy, instead of multiple levels as in SPTs (see Figure 4).
- SPTs evaluate the view distance r for each node in a vertex program to allow culling nodes depending on their actual distance. For example, nodes further away could be rendered with a coarser level of detail. While this would also be possible for MOSPTs, there is no benefit in doing so, since culling a vertex in the vertex shader does not reduce its rendering time. Therefore, we just let the GPU process all nodes with $r_{max} < min(r)$ (see Figure 4).
- There is no need to cull any node with $r_{min} > r$ (i.e., interior nodes), since these nodes also form part of the more refined nodes and should therefore be rendered in any case.
- Again due to the constant d for each MOSPT level, it is not necessary to save r_{max} for each node in the hierarchy on the CPU. Instead, it is sufficient to store in an *index array* for each hierarchy level the number of points contained in this and all preceding hierarchy levels. The r_{max} values for each hierarchy level can easily be recomputed on the fly.

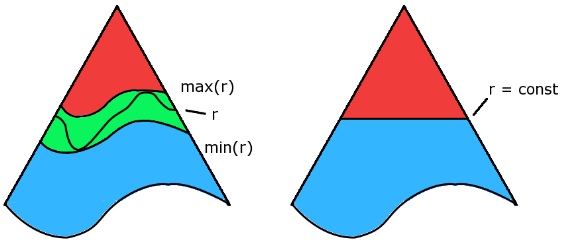

Figure 4: *Left: In an SPT, $r = const$ selects different levels of the hierarchy. Furthermore, nodes above and below r need to be culled. Right: In an MOSPT, $r = const$ selects exactly one level in a hierarchy from an MOSPT due to the screen splat error metric. All nodes above $r = min(r)$ are rendered.*

These consequences of using MOSPTs lead to a very simple rendering algorithm:

1. For each frame, the CPU calculates $min(r)$ from the bounding sphere of the model as in SPTs.
2. This is successively compared to $r_{max_i} = d_i/\varepsilon$ for each hierarchy level i until $r_{max_i} > min(r)$.
3. The number p of points to render is looked up in the index array at position i.
4. The GPU is instructed to render the first p points of the MOSPT.

Note that no vertex program is necessary to render an MOSPT. The difference between a linearized hierarchy used for an SPT and a linearized hierarchy created for an MOSPT can be seen in Figure 3. The SPT hierarchy contains additionally created points for the inner nodes. For the SPT, all points from the root node down to nodes that are just visible from the current viewpoint are sent to the GPU, and the inner points of the three upper levels will be culled. For the MOSPT, the same points are sent to the GPU, but all points will also be rendered, because they are part of the model at any level of detail.

4.6. MOSPT Creation

The hierarchy chosen for MOSPTs is an octree with a user-specified maximum recursion level. In a first step, an octree with empty interior nodes is created. Points are inserted into the octree consecutively and filtered down the hierarchy. There are three possibilities when a point reaches a leaf cell:

1. The leaf cell is empty, and the point is stored there.
2. The leaf already contains a point. Then the leaf is split, and both points are filtered further down the hierarchy.
3. The leaf is at the maximum recursion level and already contains a point. There are two leaf node strategies:
   a. Simply add the point to the node.
   b. Reject the new point, as it does not add any new information to the hierarchy.

In relation to nested octrees (see Section 5), the first leaf node strategy will be used for inner nodes where the number of input points is below a certain threshold. This guarantees that all input points are stored in the MOSPT. The second leaf node strategy will be used for other inner nodes, with the result that each leaf node stores exactly one point.

In a second step, representatives for the inner nodes are chosen in a bottom-up fashion. For each inner node, we choose the node whose color has the least distance to the average color of the child nodes. The chosen child node is then deleted. Finally, the octree nodes are sorted by r_{max} and stored in a sequential array.

5. Nested Octrees

5.1. Motivation and Definition

MOSPTs are an efficient data structure for rendering a large number of points. However, they face three major problems:

1. There is no way to do view-frustum culling, e.g. for inside views of the model.
2. Only one level of detail can be selected for the whole model.
3. It is not possible to render models that do not fit into the available memory.

It is well known that the first two problems can be solved by a straight-forward combination of octrees and SPTs, where only the lower levels of the octree are sequentialized, and the upper levels are used for culling and indexing. This would, however, require most of the SPTs to reside in graphics card memory, since it is unlikely that a whole SPT projects to less than a pixel. A non-sequentialized, classical octree on the other hand would be too slow because of the inadequate use of the graphics API. Therefore, we need a data structure that organizes chunks of points that are large enough so that they can be efficiently processed by the graphics hardware and streamed into memory by a single disk access, and small enough that they allow fine-grained view-frustum culling and memory control.

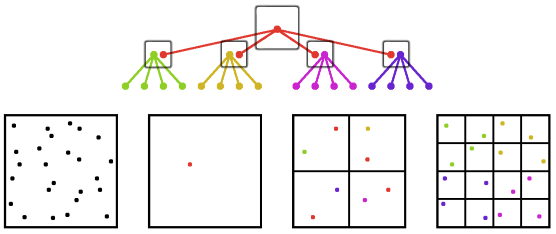
Figure 5: *2D example: a nested quadtree of three levels. Inner quadtrees are in color and limited to a depth of two.*

For this, we introduce *nested octrees*, a data structure that consists of an outer octree and nested inner octrees. The outer octree allows view-frustum culling and out-of-core rendering with incremental refinement, while MOSPTs are used as inner octrees for highly efficient rendering and API use. The main novelty in our nested octree approach is that the inner octrees overlap in the space they occupy. This overlap provides an efficient LOD representation at each level of the outer octree, so that more detailed MOSPTs are only loaded from external memory when required: refining a level in the outer octree only adds one additional level of points to the representation. Each node of the outer octree holds exactly one inner octree, which in turn holds the actual points. Each inner octree has a maximum user-specified depth, as does the outer octree. The result is a *layered* structure similar to layered point clouds [GM04], but which does not depend on a uniform sampling density due to the different construction process.

Figure 5 shows the 2D case, a nested quadtree. The outer quadtree is represented by the boxes, and is used as traversal hierarchy to reach the inner quadtrees. The overlapping inner quadtrees are limited to depth 2. Figure 6 shows the 1D case, a nested binary tree, with inner binary trees with a depth of 3 (the complete outer tree is not shown).

5.2. Hierarchy Creation

As input data for creating nested octrees we take an arbitrary point cloud possibly with color information. The depth for the inner octrees should be set neither too small (to avoid a too large number of inner octrees) nor too large (as view-frustum culling would becomes less efficient). We perform

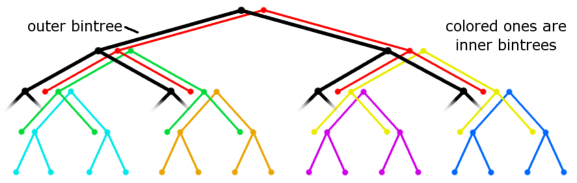

Figure 6: *1D example: a nested bintree of 5 levels. The inner bintrees (in color) are limited to depth 3.*

several sequential passes over the input, keeping a good working set in main memory during each pass. In a preliminary pass, each input file is scanned to build the bounding box of the whole point cloud from the point coordinates. This bounding box is then inflated to a cube and forms the root node R of the nested octree. R is set to be the current node, and the original point cloud set to be the current input file.

Each subsequent pass performs the following steps until all points are filled into the nested octree:

1. Create a new empty MOSPT for the current node.
2. Set its leaf node strategy to “reject” (see Section 4.6).
3. Create a “rejection file” for each child node of the outer octree, named using a unique node identifier (e.g., “R057”).
4. Scan the current input file and insert all points into the new MOSPT. Write each rejected point into one of the 8 rejection files, depending on its position.
5. If the number of points in any of the rejection files is smaller than a threshold:
   a. Set the leaf node strategy to “add,” so that points are added to instead of rejected from the MOSPT.
   b. Add the points from the small rejection file to the current MOSPT.
6. Write the current MOSPT to disk.
7. Select a non-processed rejection file, set the current node from the filename and start another pass with this rejection file as input.

Finally, the outer octree, which stores for each node just the filename of the associated MOSPT, is written to disk. As an optimization, the above algorithm can be extended to fill several levels of the outer octree simultaneously. In this case, instead of writing rejected points to a rejection file, they are inserted immediately into an appropriate MOSPT. Although the hierarchy creation theoretically requires $O(\log n)$ passes, this number is very low in practice. For the 262M points cathedral model shown in Section 6, we used 18 total octree levels with a MOSPT depth of 7, resulting in 11 outer octree levels. On our 1GB machine, we were able to compute 3 levels simultaneously per pass, so that only 4 passes were needed.

5.3. Rendering

For rendering a nested octree, the user sets the maximum number of points (the *budget*) to render for each frame to guarantee interactive navigation. For example, a budget of 12M points will keep frame rates above 10fps on a graphics hardware that processes 120M points/s. The goal of the rendering process is to render the most important points as efficiently as possible. These points are those that are: not view-frustum culled; not contained in a node that falls below a 1-pixel threshold; and are currently loaded into graphics memory.

The outer octree is loaded completely into memory and traversed with the help of a *priority queue*, with the size of the projected node bounding box as priority. There is a second *render queue* that collects MOSPTs that are to be rendered. The MOSPTs are not rendered immediately from the priority queue because the processing of child nodes can change the rendered splat size of the parent node.

For each node that is popped from the priority queue, the following steps happen:

1. Check whether rendering the current MOSPT would exceed the budget. If yes, stop traversal.
2. View-frustum cull the node. If not visible, skip node.
3. Check whether the projected bounding sphere of the *lowest level node* of the associated MOSPT is below a threshold (typically 1 pixel). If this is the case, skip the node.
4. Check whether the associated MOSPT is loaded into graphics card memory.
   - If yes, put the node on the render queue, and put its 8 children on the priority queue.
   - If not, request the MOSPT from external memory and skip it for this frame.

The splat size used to render MOSPTs is determined in each outer octree node by the lowest level descendant (more specifically, the smallest projected bounding box of its associated MOSPT) that has all requested children loaded. This prevents gaps caused by missing nodes. The splat size of leaf nodes will coarsely depend on the sampling density using this construction. Alternatively, a fixed node size can be used for rendering leaf nodes.

When the priority queue has been fully traversed, the rendering queue is traversed and all contained nodes are rendered using graphics hardware.

View-frustum culling is done in clip-coordinates. The bounding boxes of the cells can be calculated during rendering in a way that only requires additions, as described in [DD04].

Handling out-of-core requests for MOSPTs that are not in graphics card memory happens in a separate thread so that rendering can continue undisturbed. Each MOSPT is stored in a file that can be loaded directly into a graphics card buffer object without preprocessing. Even though there may be a large number of MOSPT files (about 30K files for a model with 262M points), the operating system provides fast access to the individual files (e.g., the NTFS file system uses B-Trees for large directories). The MOSPTs are managed in two LRU caches, one in graphics card memory and one in main memory.

6. Results

We demonstrate Instant Points on a point cloud of a large cathedral, which consists of more than 262M points from 77 scan positions. The accompanying video shows an interactive session where we set a minimum target frame rate of 10 frames/s, which is met or exceeded during the whole interaction. The video also shows that some areas are successively refined as they are streamed in from hard disk. The whole dataset for this animation requires 4GB on harddisk and was created automatically in about 2 hours. Instant Points makes efficient use of current graphics hardware: we achieve a throughput of 105-115M points/s during the whole animation, which is near the maximum point throughput (116M points/s) of our graphics card. Only when the viewpoint is moving fast through the model, the throughput drops to roughly 80M vertices/s. During rendering the cathedral, a maximum of some 400 rendered cells is never exceeded when using 7 levels for the inner octrees, which makes setup times for the graphics card buffer objects during rendering negligible.

These results were generated on an Intel Pentium4 3.2GHz computer with hyper-threading enabled, with two 10,000RPM harddisks (set up as a RAID 0), and 1GB of RAM. The graphics card was an NVIDIA GeForce 6800GTO with a maximum point primitive throughput of 116M points/s according to NVIDIA (we were able to confirm this maximum throughput in our own tests). All scans used in our tests were obtained using a Riegl LMS Z420i laser scanner with a range of up to 800m at an accuracy of the depth measurement of 1cm.

To compare the efficiency of the original SPTs and MOSPTs on unprocessed point data, we used one single scan of the cathedral. The original SPT contained 10,021,473 points, requiring 243MB memory, whereas MOSPTs required only 6,609,305 points (i.e., the original ones), at 107MB, which corresponds exactly to our projected total overhead of 125%. Figure 7 shows two more scenes with very similar overheads, confirming the empirical branching factor of $\alpha = 3$ for scanned datasets.

Hierarchy creation times ranged from 14 minutes for a model of 26M points (8 scan positions, with 48,793 MOSPTs of depth 5), up to slightly over 2 hours for the whole cathedral model (77 scan positions, with 33,887 MOSPTs of depth 7). These creation times are still reasonable even for usage in scanning campaigns, where day-to-day planning has to take into account the already scanned positions, and interaction with those positions is crucial.

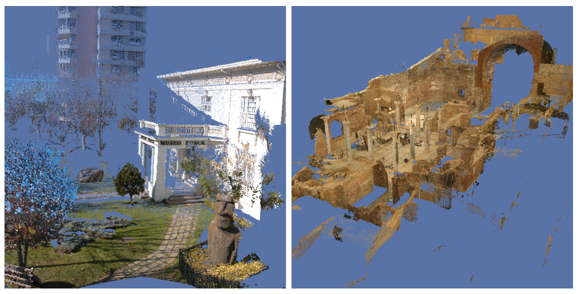
Figure 7: *Single scans of and island location ($M = 59\%$) and of an archaeological excavation site ($M = 53\%$).*

The image quality of the rendering can naturally not compete with fully postprocessed point cloud models. The goal of this work is not to provide the highest quality rendering—which is simply not possible with unprocessed point clouds—but to provide quick interaction with huge point clouds by exploiting the fact that unprocessed point clouds are simpler in structure. In the case of a high point density, the sub-sampling approach implemented by MOSPTs provides a reasonable approximation that has the advantage that no assumptions about the point cloud have to be made. Figure 8 compares a full SPT calculated using averaging with an MOSPT using subsampling. Another property of the algorithm is that it does not take the direction from where the sample was taken into account. It is therefore not possible to distinguish between front- and backfacing geometry, as is evident in the cathedral video. This is a topic for future work.

The out-of-core process inherently also leads to popping artifacts when new information is streamed into memory to refine the model. However, these popping artifacts are only temporary, and when the viewer remains stationary for a short time, he will be able to observe the full quality model when all the points are loaded. Note that not all available points will be loaded by the system, but only the amount of points that result in the desired target frame rate. Figure 9 shows screenshots from the walkthrough where the image quality of the Instant Points algorithm can be observed.

7. Conclusions and Future Work

We have presented Instant Points, a system to render huge unprocessed point clouds with only little preprocessing. The algorithm does not rely on normal vector or splat size estimation and can therefore render models with strongly varying densities and many undersampled areas, which occur often in 3D range scanner data. This is becoming a more and more important topic, since interaction with such models is often necessary already before lengthy postprocessing to fix the model or even manual mesh creation can take place. The system consists of an out-of-core data structure called nested octrees, and utilizes an improved version of sequential point trees called MOSPT, which take advantage of the restrictions of unprocessed point clouds and require less memory and render faster than SPTs.

In terms of future work, we want to investigate more advanced methods to adapt the splat size when zooming into the model, which is difficult in the absence of neighborhood information. In the realm of triangle rendering, the randomized z-buffer approach [WFP*01] handles huge polygonal scenes, and it would be interesting to compare their randomization techniques to ours. Another acceleration technique commonly used in triangle rendering is occlusion culling. While the scene structure even of huge scanned datasets like the cathedral is not necessarily amenable to culling, integrating the coherent hierarchical culling algorithm proposed by Bittner et al. [BWPP04] into the render queue of nested octrees seems straightforward. Finally, the memory savings obtained by MOSPTs need to be contrasted with dedicated compression techniques, which can achieve much higher compression rates at some additional cost (e.g., using quantization and delta-coding [KSW05]). We will investigate how these two complimentary approaches can be combined while maintaining a high rendering speed.

Acknowledgements

This work was supported by the TUW-Ilscan Centre of Competence at the Vienna University of Technology.

References

[BDS05] BOUBEKEUR T., DUGUET F., SCHLICK C.: Rapid visualization of large point-based surfaces. In *Proceedings of VAST 2005* (2005), pp. 75–82.

[BHZK05] BOTSCH M., HORNUNG A., ZWICKER M., KOBBELT L.: High-quality surface splatting on today's gpus. In *Eurographics Symposium on Point-Based Graphics* (2005), pp. 17–24.

[BWPP04] BITTNER J., WIMMER M., PIRINGER H., PURGATHOFER W.: Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum 23*, 3 (2004), 615–624.

[DD04] DUGUET F., DRETTAKIS G.: Flexible point-based rendering on mobile devices. *Computer Graphics and Applications 24(4)* (2004), 57–63.

[DVS03] DACHSBACHER C., VOGELGSANG C., STAMMINGER M.: Sequential point trees. *ACM Trans. on Graphics 22*, 3 (2003), 657–662.

[GM04] GOBBETTI E., MARTON F.: Layered point clouds. In *Eurographics Symposium on Point-Based Graphics* (2004), pp. 113–120.

[KSW05] KRÜGER J., SCHNEIDER J., WESTERMANN R.: Duodecim - a structure for point scan compression and rendering. In *Eurographics Symposium on Point-Based Graphics* (2005), pp. 99–107.

[LE97] LUEBKE D., ERIKSON C.: View-dependent simplification of arbitrary polygonal environments. In *Proc. ACM SIGGRAPH 97* (1997), pp. 199–208.

[PSL05] PAJAROLA R., SAINZ M., LARIO R.: Xsplat: External memory multiresolution point visualization. In *Proceedings IASTED International Conference on Visualization, Imaging and Image Processing* (2005), pp. 628–633.

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

[RB93] ROSSIGNAC J., BORREL P.: Multi-resolution 3D approximations for rendering complex scenes. In *2nd Conf. on Geometric Modelling in Computer Graphics* (1993), pp. 455–465.

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

[TKDS05] T. K. DEY G. L., SUN J.: Normal estimation for point clouds : A comparison study for a voronoi based method. In *Eurographics Symposium on Point-Based Graphics* (2005), pp. 39–46.

[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. ACM SIGGRAPH 2001* (2001), pp. 361–370.

[WGK05] WAHL R., GUTHE M., KLEIN R.: Identifying planes in point-clouds for efficient hybrid rendering. In *13th Pacific Conference on Computer Graphics and Applications* (2005).

[XC04] XU H., CHEN B.: Stylized rendering of 3d scanned realworld environments. In *Proc. Symposium on Non-Photorealistic Animation and Rendering 2004* (2004), pp. 25–34.

[YSGM05] YOON S.-E., SALOMON B., GAYLE R., MANOCHA D.: Quick vdr: Out-of-core view-dependent rendering of gigantic models. *IEEE Trans. on Visualization and Computer Graphics 11*, 4 (2005), 369–382.

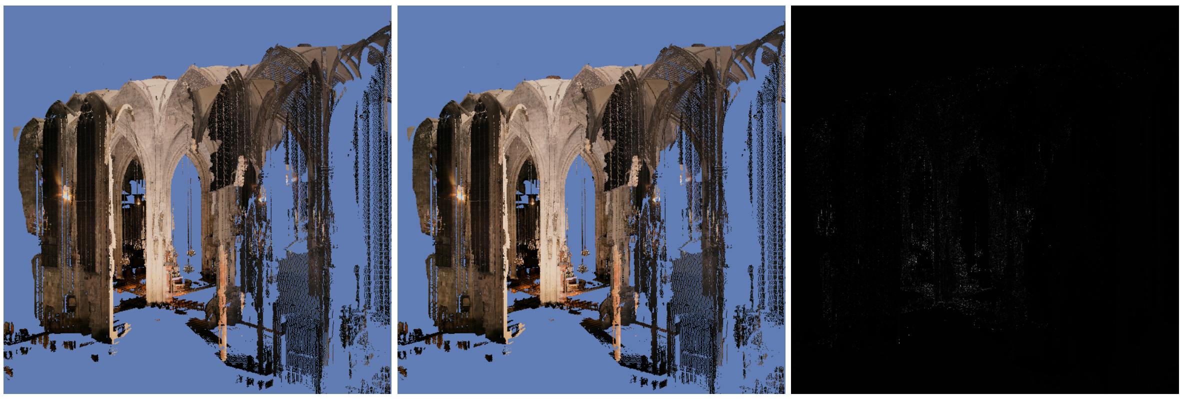

Figure 8: *Quality comparison of SPT (left) and MOSPT (middle) rendering. The colors for the SPT were obtained by averaging, those in MOSPT come from selecting representative colors from child nodes. While the difference image (right) does show some discrepancies, the order of magnitude of these differences is not higher than that of the noise contained in the original data.*

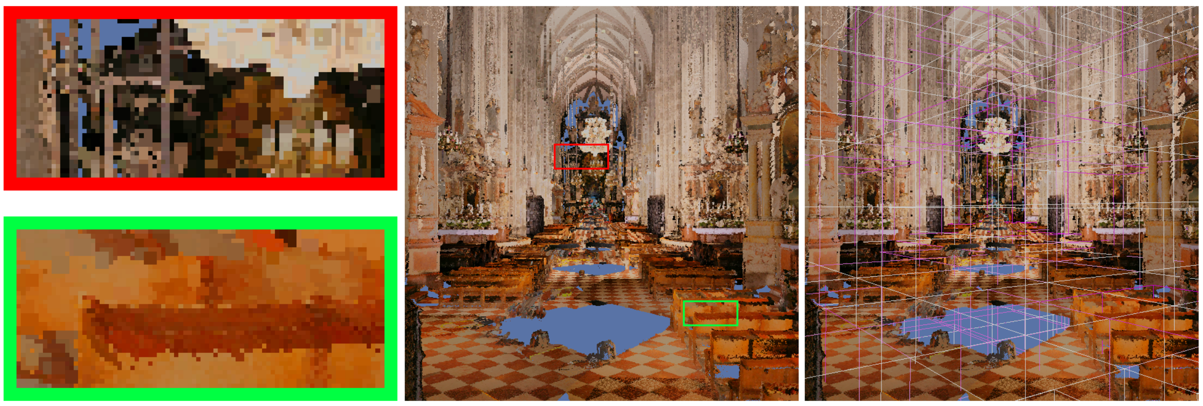

Figure 9: *Screenshots of the walkthrough in the cathedral model. The zoomed parts of the middle image show that the point sizes are similar for the far away and for the nearby regions, proving that the LOD selection works well. The middle image was rendered with 9,890,458 points in 308 cells, and loading was completed. In the right image, the magenta bounding boxes indicate that some children of these cells were not rendered because the projected sizes of their bounding boxes were too small.*