TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

D I P L O M A R B E I T

Hardware-Accelerated Rendering of Unprocessed Point Clouds

ausgeführt am Institut für

Computergraphik und Algorithmen

der Technischen Universität Wien

unter Anleitung von

Univ.Prof. Dipl.-Ing. Dr.techn. Werner Purgathofer,

und Univ.Ass. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

als verantwortlich mitwirkendem Assistenten

durch

Claus Scheiblauer

Spratzerner Kirchenweg 71

3100 St.Pölten

October 16, 2006

Datum

Unterschrift

Abstract

In this diploma thesis a fast rendering algorithm for very large point clouds is described. A point cloud is simply a set of unconnected 3D coordinates in cartesian space. Each coordinate of such a set is interpreted as a point in space. A point cloud is the result of a sampling process, where either a laser scanner samples a real environment, or the data structure of some already existing graphical model is point sampled. During rendering it is attempted to reconstruct the sampled model from the given point cloud. The algorithm presented in this thesis builds on two new data structures, namely Memory Optimized Sequential Point Trees and Nested Octrees. It includes an out-of-core part, which means that it is also possible to render models that do not fit in the main memory of the computer, and an occlusion-culling part, which means that objects, which are hidden by objects closer to the viewer, do not have to be rendered. The algorithm is developed primarily for the fast rendering of point clouds, i.e., with a high frame rate, whereas the visual quality of the rendered point clouds is not in the focus of this work. The algorithm does not need any additional attributes at a point besides the position.

Zusammenfassung

In dieser Diplomarbeit wird ein schneller Renderingalgorithmus für Punktwolken beschrieben. Eine Punktwolke ist einfach eine Menge von unzusammenhängenden 3D Koordinaten im Kartesischen Raum. Jede Koordinate einer solchen Menge wird als Punkt im Raum interpretiert. Eine Punktwolke ist das Resultat eines Abtastprozesses, bei dem entweder ein Laserscanner eine reale Umgebung abtastet, oder bei dem die Datenstruktur eines bereits existierenden graphischen Modells in eine Punktwoke umgewandelt wird. Während des Renderings wird versucht das abgetastete Modell aus der gegebenen Punktwolke zu rekonstruieren. Der Algorithmus, der in dieser Diplomarbeit vorgestellt wird, basiert auf zwei neuen Datenstrukturen, nämlich auf Memory Optimized Sequential Point Trees und Nested Octrees. Er enthält einen Out-of-Core Teil und ist dadurch in der Lage Modelle zu rendern, die nicht komplett in den Hauptspeicher des Computers passen. Er enthält auch einen Occlusion Culling Teil, wodurch es möglich wird Objekte nicht zu rendern, die durch andere Objekte, welche näher beim Betrachter sind, verdeckt werden. Der Algorithmus wurde hauptsächlich für die schnelle Darstellung von Punktwolken entwickelt, das heißt für eine hohe Bildwiederholfrequenz, wohingegen der visuellen Qualität der gerenderten Punktwolken weniger Aufmerksamkeit geschenkt wurde. Der Algorithmus benötigt keine zusätzlichen Informationen zu einem Punkt ausser dessen Position.

Contents

Chapter 1

Introduction

This diploma thesis describes an algorithm for rendering point-based objects as fast as possible. The geometry of a point-based object is completely defined by points. As fast as possible means at interactive frame rates, which is anything between 1 frame per second (FPS) to 60 FPS. Real-time frame rates above 60 FPS are possible for models which do not need the out-of-core part of the algorithm. Points are not the only means for modeling objects, because geometrical models can be defined in various ways. They can either be constructed from a set of geometric primitives that can be rendered directly, e.g., triangles and lines, or they can be defined implicitly by mathematical equations, e.g., quadrics and splines. As Levoy and Whitted have shown, any geometrical model can be converted to a point-based model and can be rendered as a continuous object to screen [LW85]. So points can be seen as the most basic rendering primitive, and converting a geometrical model to a point-based model is always possible. The other way, which is converting a point-based model to a representation consisting of more complex primitives, is not straight forward, as it can be difficult to find the surfaces that were sampled for a point-based model. Various techniques exist that can be used to reconstruct a connected model from a point-based one [SS05].

1.1 Point Definition

A point in the mathematical sense is the intersection of two straight lines. It can be interpreted as a zero-dimensional entity which has only a position. It has no extent and would therefore be invisible. Since an object consisting of points as defined above would be rendered invisibly, a different definition is given.

A point used for rendering has a position and optionally other attributes, and has an extent of at least one pixel on screen when visible from the current camera position.

The word point in this diploma thesis always means “a point used for rendering”, and is described at the minimum by a 3-tupel,

$$Point = (x, y, z), \tag{1.1}$$

where x, y and z are coordinates in a cartesian coordinate system. If the point is colored, it is described by a 6-tupel,

$$Point\ with\ Color = (x, y, z, r, g, b), \tag{1.2}$$

where r, g, and b are the values of the red, green, and blue color channel respectively. Note that no alpha value is used, as only solid point samples are used.

1.2 Why Use Points for Rendering?

Any geometrical model that should be rendered must first be converted to some rendering primitive, that is a geometrical primitive which can be sent through the rendering pipeline. When using a software-based rendering pipeline, every stage is under the control of the programmer and can be optimized to the needs of the available models. One advantage of point-based models is that the setup process for triangles can be skipped during rendering. The setup process for triangles includes for example polygon clipping, scan conversion, and texture mapping. These things are time consuming considering a software-only implementation of the rendering pipeline. Today consumer graphics cards are ubiquitous, and they provide hardware accelerated rendering pipelines. This means that the setup process for triangles is handled in hardware, and the pure rendering-speed advantage of point-based models is therefore fading away. In fact, the number of point primitives that can be rendered per second on the current generation of graphics cards is the same as the number of triangles that can be rendered per second. From the original idea of Levoy and Whitted [LW85] of using points as some kind of 3D texture by converting all geometrical models to points, the application areas of points as rendering primitives have shifted.

1.3 Application Areas of Points as Rendering Primitives

Today with the wide availability of the hardware-accelerated rendering pipeline on graphics cards, the conversion from triangle-based models to point-based models does not promise much of an advantage, since the graphics cards are tuned to rendering triangle-based models as fast as possible. The application areas for point-based models are where the geometric models originally consist of points, or where scientific data can easily be converted to points, or where still no hardware-accelerated rendering pipeline exists, like on low-end handheld devices.

Fig. 1.1: An example of a terrestial laser scanner. Here a Riegl LMS z420i is shown. The range measuring laser is situated behind the blue windows in the scanner's casing. After taking range samples from the environment, the camera on top of the scanner takes photographs of the scanned environment.

1.3.1 Visualization of Range Scans

One area where the rendering of point-based models is of interest is the visualization of point-sampled environments. Point-sampled environments are becoming more common due to the increased availability and usage of range-scanning devices. Range scanners scan a real environment, and the output of a range scanner is a set of 3D coordinates. These coordinates are the surface points of the scanned environment. A point viewer, which is a program to visualize the coordinates on screen, interprets these 3D coordinates as points for rendering. The interpreted 3D coordinates are referred to as *point cloud*. If the scanning resolution is high enough, the point viewer can reconstruct the environment from the output of the range scanner.

The difficulty when rendering the point clouds directly from a range scanner is the lack of information between the points. The geometry of the model is only

known at the sampled points. Admittedly the color, coming from the photographs taken by the camera mounted on top of the scanner, is also available for the area between the samples. Apart from converting the whole model into a triangle mesh, which is tedious and time consuming, it is theoretically possible to apply a texture to each point, but then the points would have to be defined with a radius to describe their extent in space. Furthermore a normal would also be required, to orient neighboring points along the surface they describe. But if no normal is available, the texture cannot be utilized in a meaningful way. Today point clouds from a range scanner usually store only one color per point.

When the camera of the point viewer is in some distance to the point cloud, then the projected points will probably result in a continuous image of the scanned model on the viewplane. The continuous image becomes undersampled when the viewpoint moves away from the model. To accommodate for this, either the image in screen space could be filtered, or the points could be rendered with filtered textures to avoid aliasing. When the camera moves towards the point cloud and the resolution of the scanned model does not allow for a continuous image on the viewplane any more, a gap-filling strategy has to be used to simulate a solid model.

In Figure 1.1 a range scanner is shown. The mode of operation of a range scanner is described in the following: The distance to obstacles in the environment is measured by the time of flight of a laser pulse. Figure 1.2 depicts a situation where a scanner is situated in the environment. The laser scanner takes discrete samples on one vertical line, then rotates around its vertical axis and scans another line. The angle between two consecutive samples taken in both directions can be chosen by the user. The scanner is able to scan a 360 degrees vertical frustum around it in one run, and afterwards it takes pictures from the scanned environment with the camera mounted on top of the scanner's casing. The pictures are later used to colorize the taken samples. The point clouds are not colorized by the scanner automatically, but by an external program.

1.3.2 Scientific Visualization

Another area where point-based models are of interest is the visualization of scientific datasets. In such cases like volume rendering, points are well suited to represent the data. Isosurfaces can be extracted from volume datasets, and the measured or simulated values at the positions in space can be represented with points. An example is shown in Figure 1.3, where an isosurface is extracted from a dataset and then rendered as a point cloud.

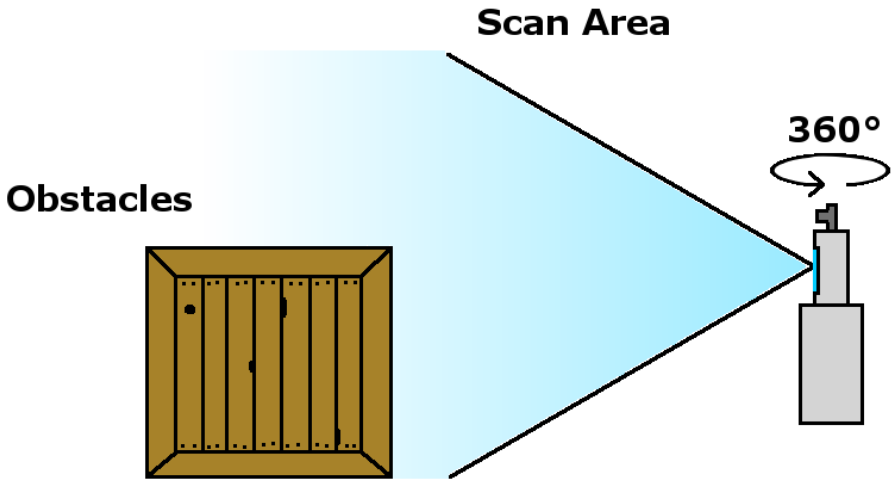

Fig. 1.2: A laser scanner scanning the environment. The total angel of rotation can be up to 360 degrees. The scan area depicts the limitation of the laser measurements in vertical direction. After the range scan, photographs of the scanned environment are taken by the camera on top of the scanner's casing.

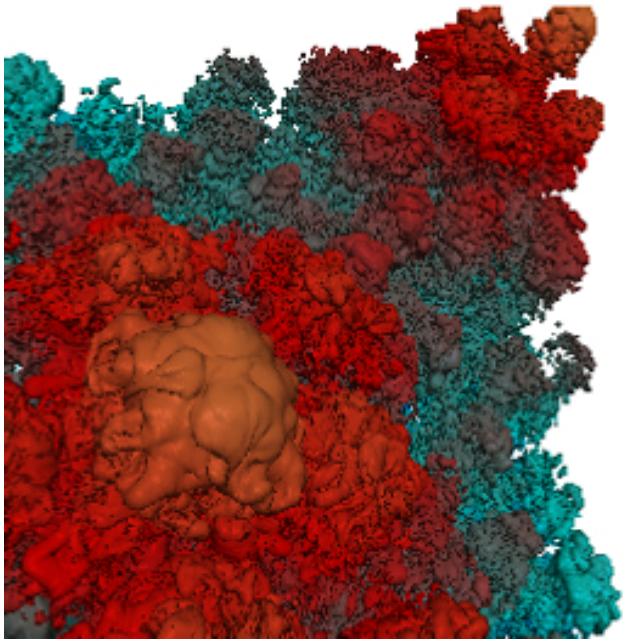

Fig. 1.3: Closeup view of an isosurface feature in the mixing interface of two gases for a simulation of a Richtmyer-Meshkov instability in a shock tube [MCC+99] as rendered with the Layered Point Clouds algorithm [GM04].

1.3.3 Rendering Models on Handheld Devices

The third area where point-based models are useful, are handheld devices like PDAs, which do not have a dedicated graphics processor. Such devices mostly have screens with about 240x320 pixels resolution and have to use a software-based rendering pipeline. Due to the low screen resolution, triangles will often be projected to only one pixel, so the gain of triangle-based models that can take advantage of rasterization coherence is not existent. Triangles also need to be set up for rendering, which needs time. Such devices also have only limited memory, and a point-based model can be efficiently compressed. All these things make point-based models very well suited for handheld devices. Efficient algorithms exist that can render point-based models at interactive frame rates on PDAs [DD04].

1.4 Contribution

Most point-rendering algorithms have certain requirements to work properly. Normals are always assumed to be available, and the models should also be uniformly sampled. There exists no dedicated algorithm for rendering unprocessed point clouds. This is where the contribution of this diploma thesis comes in, as we present the first point rendering system with minimal requirements, and which does not demand for postprocessing of point clouds and at the same time renders enormous amounts of unprocessed points at interactive rates with negligible preprocessing. With unprocessed point clouds we mean point clouds that are not interpreted in any way, i.e., no triangulation or normal estimation has been performed. A point in an unprocessed point cloud is only defined by a 3D position. It can optionally have an RGB color, if the color is available from the scanning process.

Having only a limited amount of attributes available per point, the visual quality of the rendered models cannot compete with models that are postprocessed and rendered with a high-quality splatting algorithm. So the main contribution of this thesis is to show how to trade this reduced image quality against significantly increased visualization speed and improved memory requirements. We introduce two new data structures, which are

- Memory Optimized Sequential Point Trees and
- Nested Octrees.

The Memory Optimized Sequential Point Trees (MOSPTs) build on the Sequential Point Trees (SPT) [DVS03]. A MOSPT is a sequentialized hierarchy that can be traversed completely on the GPU and requires only the original points for

providing a level-of-detail mechanism. The memory requirements are decreased by more than 50% compared to the SPT data structure.

The Nested Octree data structure is used as an outer hierarchy, which holds a MOSPT at each node. The build up algorithm of the Nested Octree is especially well suited for the processing of many point clouds which come directly from a range scanner. The only thing the user has to do for each point cloud, is to define the region where the important points are. From then on the build up process is automated. The rendering includes view frustum culling and an out-of-core part, so models that do not fit completely in the main memory of the computer can be shown. During rendering, also occlusion culling is performed, which tries to find areas in the model that are hidden, and then these hidden areas do not have to be rendered. The system maintains interactive framerates at any time, and the user can set a limit for the minimum rendered frames per second.

1.5 Overview

This is the overview of the content of the following chapters.

- Chapter 2 gives an overview of existing rendering algorithms for point clouds. The algorithms reviewed here concentrate on fast rendering or out-of-core rendering of point clouds.
- Chapter 3 explains the Memory Optimized Sequential Point Tree data structure, which is needed for rendering point clouds as fast as possible. The presented data structures are effective for point clouds that fit in the memory of the graphics card.
- Chapter 4 explains the Nested Octree data structure, needed for out-of-core rendering of point clouds. With this data structure it is possible to render point clouds that do not fit in the main memory of a PC.
- Chapter 5 describes special problems that occurred during the implementation of the rendering algorithm and the build up algorithm.
- Chapter 6 shows the results of the point-rendering algorithm on large and small models and demonstrates the effect of the out-of-core strategy.
- Chapter 7 finally summarizes the presented data structures and algorithms, and also gives an outlook to possible future enhancements.

Chapter 2

Previous Work

In recent years points have become increasingly popular as rendering primitives. From the seminal work of Levoy and Whitted [LW85] on points as rendering primitives, it took more than ten years until Grossman and Dally presented their implementation of a point-rendering system [GD98]. Levoy and Whitted suggested to use points as a universal meta primitive. Their idea was to convert all geometric objects to points, so that the rendering pipeline could be the same for all geometric objects, independent of the way the objects were modeled. They also proved that it is possible to render any geometry consisting of points as a continuous object on screen. The point-rendering system of Grossman and Dally showed, that arbitrary models consisting only of points can be rendered in real time. They also introduced visibility cones for testing the visibility of several points at once, and they introduced incremental block warping, which means that blocks of points are warped to image space using incremental calculations.

The development of point-based rendering algorithms can be divided in two different areas. In one area the development leads towards ever higher quality surface rendering algorithms, and in the other area the development is more concentrated on the fast rendering of point clouds but there exists no definite distinction between the two areas. The main focus of this thesis is on the fast rendering of point clouds, whereas the visual quality of the rendered models will not be considered as important.

2.1 High-Quality Point Rendering Algorithms

In the year 2000 Pfister et al. [PZvBG00] presented an algorithm which improves the rendering quality of point-based models compared to previous methods [LW85, GD98], by introducing visibility splatting and surfel mipmapping. They give the definition of a *surfel* as a zero-dimensional n-tuple with shape and shade attributes that locally approximates an object surface. The additional informations, like normal and radius, that are stored at a surfel, are necessary for aligning the surfels during rendering along the surface they describe. With visibil-

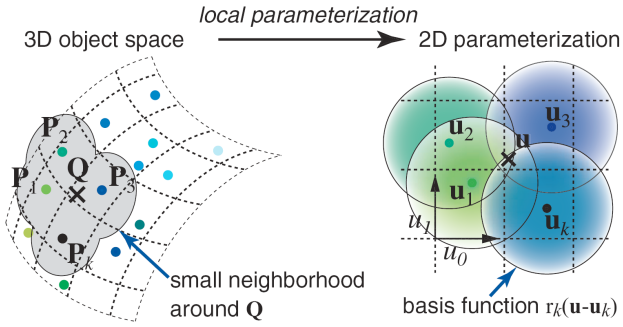

Fig. 2.1: A continuous texture function is locally approximated by basis functions [ZPvBG01].

ity splatting the depth buffer is established in a separate rendering pass before the color buffer, so that visibility calculations are separated from the lighting calculations. The points then receive colors which are the result of a linear interpolation of the appropriate texture samples, similar to texture mipmapping, therefore this process is called surfel mipmapping. The texture samples store only one color at each mipmap level. The surfels algorithm is a fast and high-quality rendering algorithm for point-based models.

In 2001 Zwicker et al. [ZPvBG01] formulated a screen space Elliptical Weighted Average (EWA) filter [Hec89], which makes EWA texture filtering available to point-based rendering techniques. The difference to previous approaches for texturing point-based models is, that EWA filtering actually reconstructs, bandlimits, and resamples the texture during rendering, which results in superior antialiasing. Previous point rendering systems used prefiltered textures, and antialiasing could be done with supersampling or filtering the already rendered images. In their paper they present the mathematical formulation of screen space EWA texture filtering for irregular point data. To make EWA texture filtering possible, each surfel needs a position and normal and is associated with a radially symmetric basis function. Further it needs coefficients that represent continuous functions for red, green, and blue color components. As basis functions they use elliptical Gaussian functions. The coefficients are computed during a preprocess by sampling a continuous texture. Any point on the surface of a point-sampled object can then be textured by summing up the contributions of the weighted basis functions of a small neighborhood around the point. Figure 2.1 shows the contributions of different basis functions for a point Q on a surface. In Figure 2.2 a one-dimensional signal reconstruction is shown, as it is used by EWA texture filtering.

After warping the basis functions to screen space and then bandlimiting them, they are referred to as reconstruction kernels. The functions used for the ban-

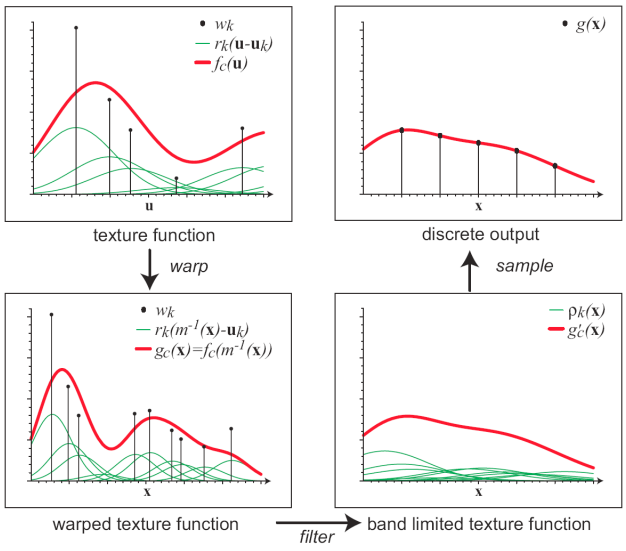

Fig. 2.2: Here in the one-dimensional case the warping, bandlimiting, and resampling of a continuous function from basis functions is shown [ZPvBG01].

dlimiting filter and the basis functions are both elliptical Gaussian functions, so it is possible to express the resampling kernel as a single Gaussian function. During rendering the reconstruction kernels are evaluated, and their contributions are added up for each pixel and normalized, which finally results in an antialiased texture for the point-sampled model. This method is called surface splatting. It provides high-quality texture mapping for point-based models.

Several papers from then on improved high-quality point rendering based on the ideas of Pfister et al. [PZvBG00] and Zwicker et al. [ZPvBG01]. With the advent of vertex- and pixel shaders and the hardware accelerated rendering pipeline, EWA-filtered rendering techniques are now able to achieve over 20 million rendered splats on a 512x512 pixel viewport [BHZK05]. Figure 2.3 shows the different rendering passes for the algorithm presented by Botsch et al. [BHZK05]. The use of several rendering passes is necessary for high-quality splatting algorithms, but this also means that they cannot provide the maximum throughput in terms of vertices per second (VPS). The high-quality splatting algorithms need additional attributes at each point, especially normals. These additional attributes also prevent them to achieve maximum rendering rates.

2.2 Fast Point Rendering Algorithms

Rendering algorithms which try to render as many points as possible per second trade visual quality against the number of rendered points.

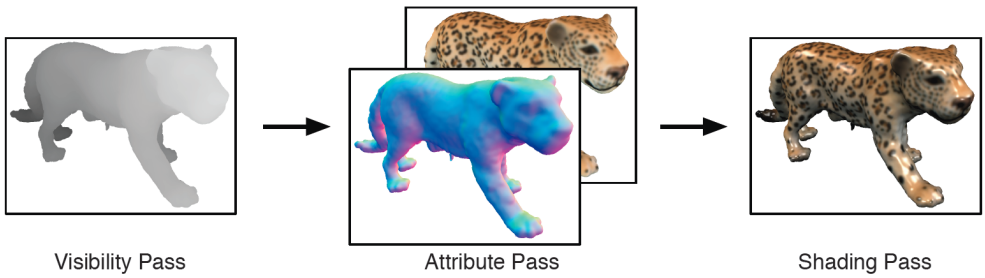

Fig. 2.3: The deferred shading pipeline for GPU-based splatting as presented by Botsch et al. [BHZK05]. The visibility pass fills the z-buffer, such that the attribute pass can correctly accumulate surface attributes, like color values and normal vectors, in separate render targets. The final shading pass computes the actual color value for each image pixel based on the information stored in these render targets.

This can be useful if the models do not provide all informations that are necessary for high quality rendering, e.g., a normal for each point, or if the sampling density is highly varying. 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 all models, e.g., long range scans, where the data is simply too sparse in many regions. Figure 2.4 shows the result of a range scan, where the area around the scanner's position is sampled densely. For areas further away, the distance between neighboring points increases. This is due to the sampling process, where the laser takes samples in discrete steps. The angle between consecutive shots of the laser remains constant, and so the distance between two consecutive taken samples is larger if the samples are at a greater distance to the scanner.

To enable normal estimation, lengthy manual postprocessing is inevitable, where the geometry is reconstructed from the available scan data, and the noise within the scan data has to be removed. Then a surface can be approximated to the available points, and from the surface the normals can be assigned to the points.

It can also be useful to maximize the number of rendered points for large models in a real-time application like a walkthrough. In this case it is more important to render many frames per second than to render them in high quality. There exist algorithms which approximate parts of the point-based models with textures like Wahl et al. [WGK05] did, or with normal-mapped polygons as shown by Boubekeur et al. [BDS05], but they have to resample the original point cloud, and so not all original points are preserved. To render very large models, an out-of-core part in the algorithm seems inevitable. An alternative is to use compression. Krüger et al. [KSW05] presented a very effective data structure for resampling

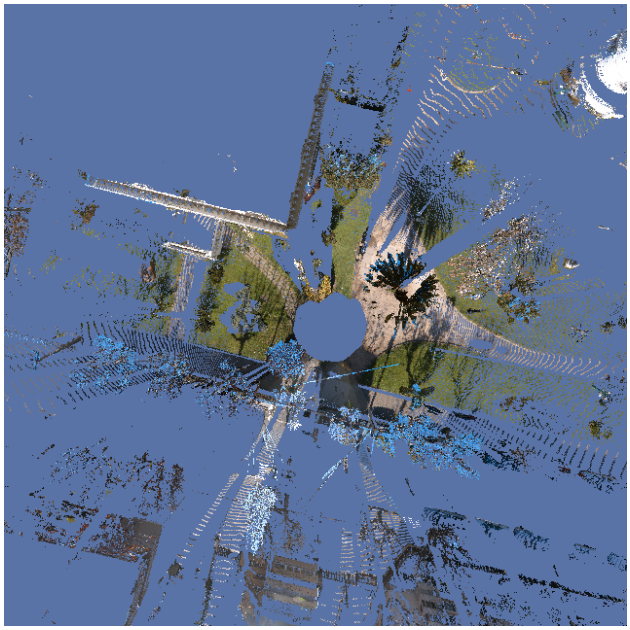

Fig. 2.4: A range scan as seen from above.

the original point cloud and managed to store models with more than 160 million points in 256MB of graphics memory. The graphics card can then be used to decompress the compact representation on the fly and store the vertices in vertex buffer objects (VBOs). In OpenGL, VBOs are used to store information in the graphics card memory. If the information is needed during rendering, e.g., the coordinates of a vertex, the GPU has fast access to it, as the information does not have to be sent over the system bus (in DirectX the VBOs are called vertex buffers). By using VBOs, they achieve 50 million VPS, which includes decoding and rendering of the vertices. It is a lossy compression algorithm, but because of this the compression is also very effective. Compression and is not yet part of the system presented in this thesis.

The following algorithms are discussed in greater detail, as they are closely related to the rendering system that was developed for this diploma thesis.

2.2.1 QSplat

The QSplat algorithm 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 sur-

rounded by its parent. The build up of the structure includes finding normals and assigning a bounding sphere to each point. If polygon models are used, it is easy to compute the normals. The radii of the bounding spheres can be derived from the distance between neighboring vertices such that no holes are left during rendering. If point samples are used as input, first a plane has to be fit to the vertices in a small neighborhood to find the normals. Then the whole model is surrounded by a bounding box, from which the bounding sphere hierarchy can be built up by further subdividing the bounding box and splitting up the contained point samples. The result of this is a binary tree which contains successively smaller bounding spheres for each level, down to the original points. A higher branching factor at the interior nodes will reduce the number of interior nodes. For better memory efficiency, interior nodes are combined to increase the average branching factor to approximately 4. 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 level-of-detail (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. If it falls short of the threshold, then the following levels of the hierarchy will certainly not contribute to the appearance of the model, and so the traversal of the following levels can be skipped. Instead a splat with the attributes of the current intermediate node is drawn.

After the bounding sphere hierarchy has been built, the attributes of the nodes are quantized. In the bounding sphere hierarchy, it is possible to encode the position and the radius of a bounding sphere relative to its parent sphere. QSplat uses 13 different values for the radius of a sphere, which express the size relative to the parent's sphere radius. So it is possible to have a radius that is $\frac{1}{13}$ up to $\frac{13}{13}$ of the parent's radius. Similar the position of the bounding sphere's center is described, only that then the distance on the X, Y, and Z axis are quantized to parts of 13. This would give 13^4 possible combinations for the attributes of a bounding sphere, but in reality only 7621 combinations are valid [RL00]. Other ones result in bounding spheres which are not completely covered by the parent sphere. The valid combinations can be encoded in 13 bits, and these are the first 13 bits for the description of a QSplat node. The normals are also quantized. Instead of saving coordinates of a normal vector, the possible directions of a normal vector are given by a 52x52 grid for each of the six sides of a cube around the point sample. This quantization suffices to produce no visible artifacts during rendering due to errors in the shading calculations. This way a normal vector can be encoded in 14 bits, and the direction can be decoded during rendering by a single table lookup. The color of a node is quantized to a 5-6-5 format, therefore it can be saved in 16 bits. The size of a cone of normals for a node is encoded in just 2 bits. The

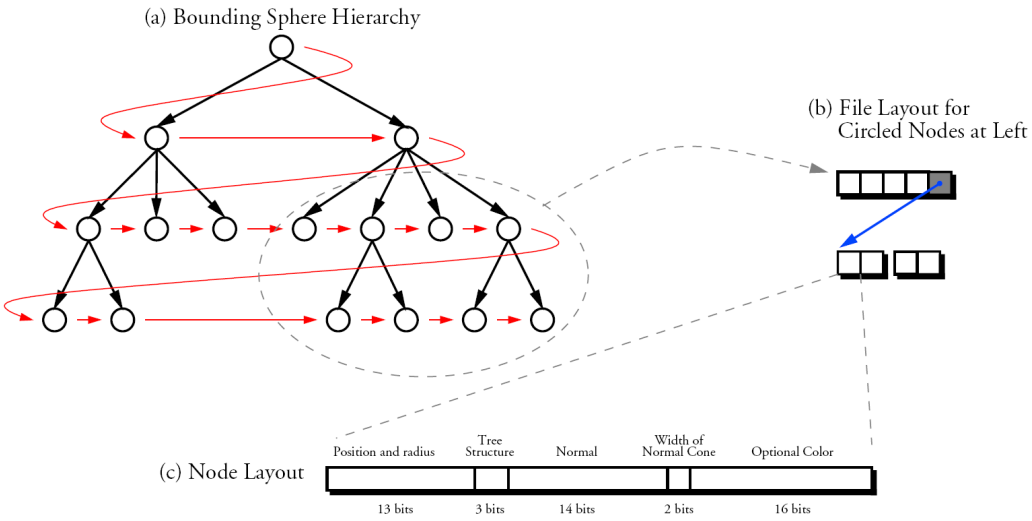

Fig. 2.5: The file and node layout of QSplat. (a) The tree is stored in breadth-first order. (b) The link from parent to child nodes is established by a single pointer from a group of parents to the first child. At leaf nodes the pointer is not present. (c) A single node occupies 48 bits.

bits describe half angles, whose sines are $\frac{1}{16}$, $\frac{4}{16}$, $\frac{9}{16}$, and $\frac{16}{16}$. The cone of normals is used for backface culling, and despite its quantization, 90 percent of the nodes which would be culled using exact normal cone widths are discarded.

The quantized hierarchy is written to disk. For rendering, the hierarchy is read into memory. At first, only visible parts are read in. This can lead to lags when the user turns to a part of the scene which is not already in memory, since then it has to be loaded from a disk. In this case, QSplat does not draw the scene to its entire detail level. This is possible because the bounding sphere hierarchy is laid out in breadth-first order. This means that the scene is represented as a whole at a certain LOD once this level has been read into memory. So if the user zooms into the scene, the next level has to be loaded, and in the meantime, the last available level in memory is drawn to the screen only with larger splats. If the user waits, the rendering will refine to the highest available detail level, as soon as it has been transferred to memory. QSplat uses splats to draw the nodes of the bounding sphere hierarchy. They are correctly sized and oriented elliptical splats. On average, the QSplat data structure with quantization applied uses 50 bits per node, and includes node position, bounding sphere radius, normal, color, cone of normals, and 3 bits of information for the traversal. To minimize the number of pointers needed for hierarchy traversal, QSplat uses at each node only one pointer to the children of this node. Furthermore, the pointer is not needed if all of the children are leaf nodes. The average branching factor for a node in the bounding

sphere hierarchy is 3.5, which means the total number of nodes will be 1.4 times the number of leaf nodes. The storage requirements for the whole tree equals approximately 9 bytes times the number of leaf nodes. A deficiency of QSplat is that the encoding of the position and radius and the hierarchical traversal require it to use the CPU for decoding the information.

2.2.2 ρ-grids

An octree can be described as a recursive grid, with a regular and uniform subdivision at each level. One octree node is subdivided into $2\,x\,2\,x\,2 = 8$ sub cells at the next octree level, such that the original cube is regularly and uniformly subdivided by the cells of the next octree level. ρ-grids are a generalization of this idea. They subdivide each node into $\rho\,x\,\rho\,x\,\rho$ sub cells.

Duguet et al. showed that a ρ-grid with $\rho = 3$ is the best compromise in terms of rendering cost and memory cost if the attributes per node do not exceed 16 bits [DD04]. Duguet et al. use a quantized normal encoded in 13 bits similar to [RL00], and 3 bits for a material index. For $\rho = 3$ a ρ-grid is dubbed trigrid. To get the actual position of a splat, the information which is implicitly encoded by the position of a cell in the trigrid is decoded. The position of the root cell is known. It is projected to window coordinates, which yields homogeneous coordinates before viewport transformation. To get the position of the child cells, their displacement vectors relative to the parent cell are precomputed, projected to window coordinates, and stored in a table. This is done for each trigrid level. Then during rendering the trigrid is traversed in depth-first order, and the positions of the child cells can be computed from the position of their parent cell with just three additions. What remains to be done is the viewport transformation.

The trigrid is a hierarchical rendering algorithm. The decision if an intermediate node can be rendered or the recursion has to step down a level is done by projecting a screen bounding rectangle. This is a conservative approximation of the projection of the bounding box of a cell. If the rectangle is larger then a certain threshold, for example larger than a pixel, then the recursion continues. Otherwise the recursion stops, and a splat is drawn with the size of the projected rectangle. This screen bounding rectangle can also be used for view frustum culling. The shading of materials is precomputed for every value of the quantized normal. This means that for every angle that can be represented by a quantized normal, the shading is available in a table. So shading can be done by a table lookup. The total bitcount for a node in the trigrid is composed of 16 bits for the attributes, and 27 bits for the childhood information. This totals 43 bits per node. The branching factor of the intermediate nodes is approximately 9, which is very efficient. Including 32 bits per intermediate node for a pointer to the node's children, the memory consumption per leaf node is approximately 4 bytes [DD04].

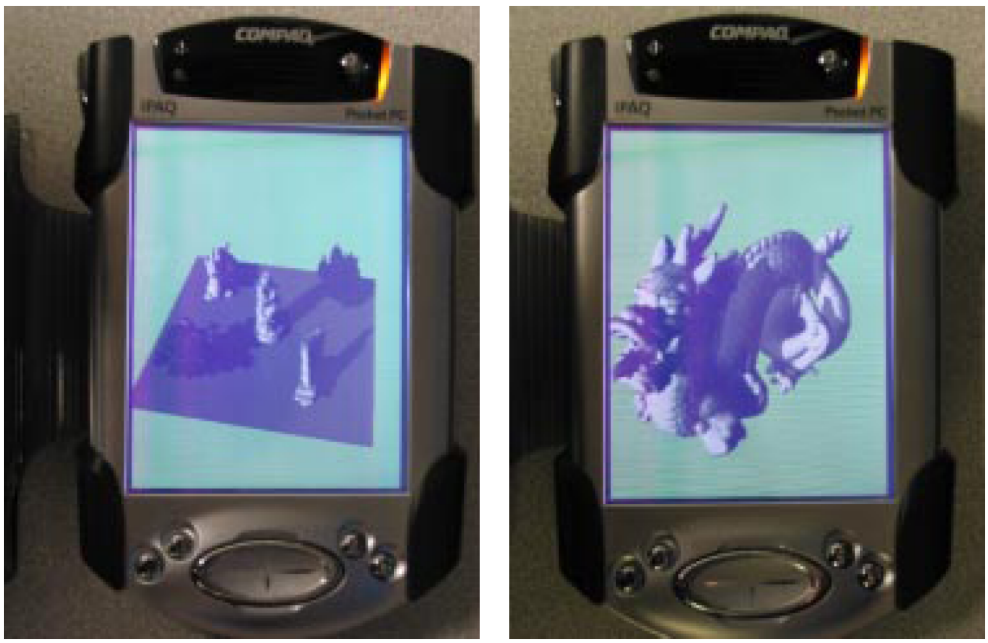

Fig. 2.6: The left image shows a ρ-grid rendering a model consisting of 1.3M points at 2.1 FPS on a 200MHz Compaq IPAQ. The model in the right image is rendered at 2.3 FPS.

We must mention though, that a trigrid is only efficient if the coordinates of the encoded point samples are clustered in three dimensions around a small area, because then the trigrid is massively occupied. If the point samples are spread over some larger area, so that the resulting trigrid is sparsely occupied, then the amount of intermediate nodes becomes dominant, and can reach up to 70 percent of the whole trigrid structure. This is not only inefficient for storage but also for rendering. A shortcoming of the structure which prevents it from using hardware acceleration is that the decoding has to be done on the CPU, since positions of the splats are dependent on the position calculation of their parent cells in the trigrid.

2.2.3 Sequential Point Trees

The SPT algorithm [DVS03] uses a hierarchical data structure, which is sequentialized so that it can be processed on the GPU. A GPU cannot resolve the hierarchical dependencies of a data structure like the CPU can, because the GPU does not have access to the whole data structure. So a restructuring is needed.

At first the points are inserted into an octree. Each point stores its center position p, an average normal n and the diameter d of a bounding sphere surrounding p. The leaf nodes are the original points, and diameters for bounding spheres around leaves should be roughly equal. The inner nodes of the octree represent

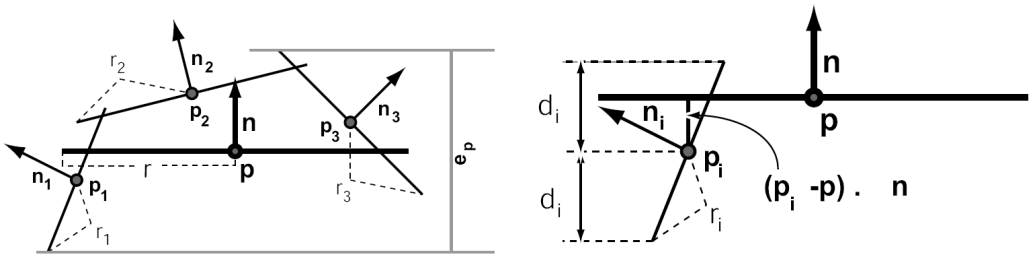

Fig. 2.7: As perpendicular error for a disk the distance between the two planes parallel to the disk enclosing all children is used.

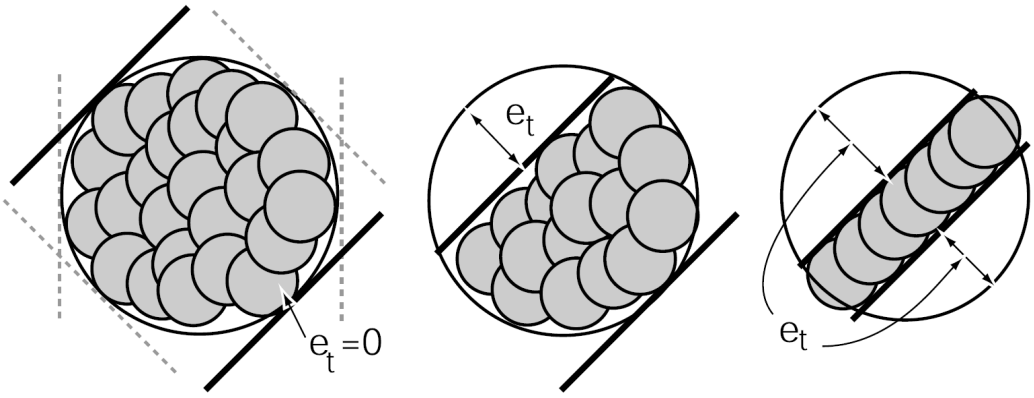

Fig. 2.8: The tangential error measures how well a parent disk approximates the children's disks in the tangent plane.

the averaged informations of their children. The points will be visualized as splats, and to calculate the informations for the inner nodes, a splat is computed that covers the splats of its child nodes [DVS03]. This results in two error measures, namely a so called perpendicular error and a tangential error. The perpendicular error measures the error when looking at the silhouette edges of an object (see Figure 2.7), and the tangential error measures how exact the parent splat covers the child splats when looking along the $-n$ direction of the parent splat onto the child splats (see Figure 2.8). The errors can be combined to a geometric error $\tilde{e}_g$. This geometric error is projected to screen space during rendering, resulting in the image error $\tilde{e}$. If $\tilde{e}$ exceeds a user defined error threshold ϵ, which is measured in pixels, then the recursion steps one level down the hierarchy. If an acceptable level is reached, a splat is drawn with size $\tilde{d} = d/r$.

The data structure so far can only be processed by the CPU. A processing by the GPU requires a different error measure. When traversing a hierarchy, it is implicitly known if an ancestor node has been rendered, because then the dependent

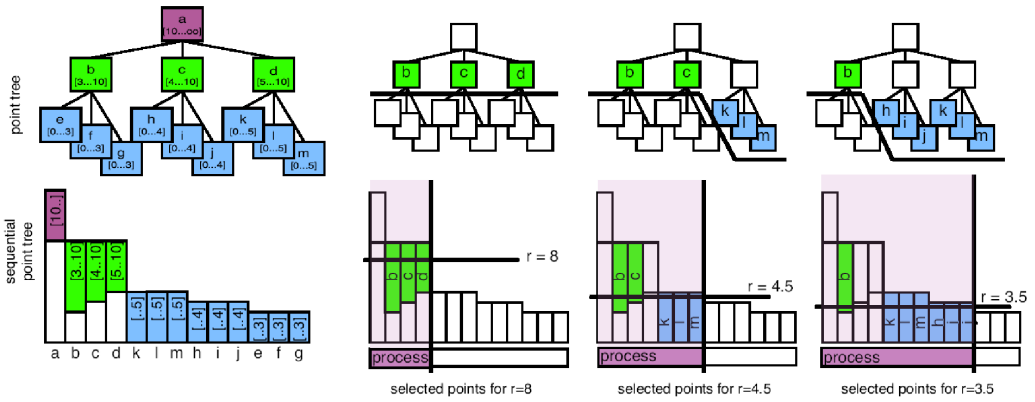

Fig. 2.9: The same nodes as hierarchical point tree and as sequential point tree. Left column: Nodes a. . .m with $[r_{min} \ldots r_{max}]$ as point tree in the upper row and as SPT sorted by r_{max} in the lower row. Other columns: In the upper row the black line shows the tree cuts for different view distances. In the lower row the cut in the SPT representation. The process bar shows which nodes are sent to the GPU, and the empty bars in the process range show, which nodes are culled by the GPU.

subtree will not even be processed. If the data structure is sequentialized, it is not known if an ancestor has already be rendered, so the algorithm has to check for this case as well. The SPT algorithm uses two simple error measures for the two cases. It calculates an r_{min} and an r_{max} for each splat, which is the minimum and maximum distance to the viewpoint for which the splat will be used. If the distance lies within, the splat will be rendered. The test in the hierarchical traversal is whether $\tilde{e} = \tilde{e}_g/r < \epsilon$, which can be rewritten as $r_{min} = \tilde{e}_g/\epsilon$. The error $\tilde{e}$ is independent of the viewpoint. The value r_{min} is stored at each node. The value for r_{max} is calculated by taking r_{min} of the parent node and adding the distance between the two nodes, which will result in a display with no holes in it. These error measures can be checked by a vertex program on the graphics card.

The r_{max} value allows for another optimization. In the left column Figure 2.9 depicts how a hierarchical point tree is represented as SPT. If the SPT is sorted by r_{max}, then the CPU can cull the 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 (this situation is shown for three different distances to the viewpoint in the bottom row on the right side of Figure 2.9). So the GPU is relieved to cull fewer nodes. The SPT is sorted by r_{max} only once in a preprocessing step. All other computations can be done directly by the GPU. The average CPU usage is only 0.4 percent during rendering, and an average of 60 Million rendered splats is achieved [DVS03]. A disadvantage of the SPT algorithm is that within an SPT, no

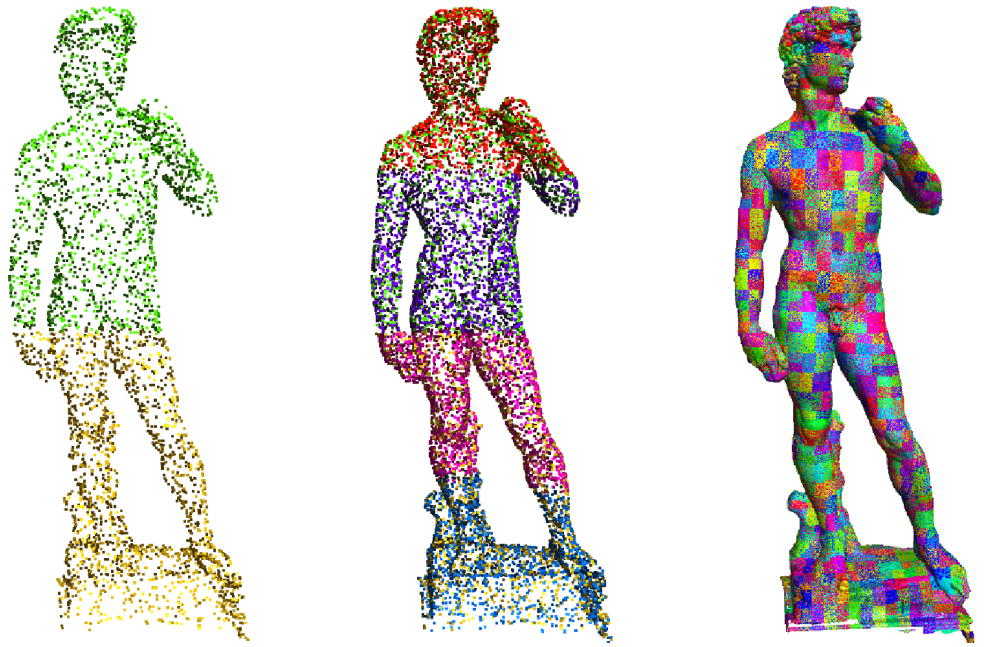

Fig. 2.10: Hierarchical rendering with layered point clouds. The figure is built from an accumulation of different point clouds. From left to right: level 0 rendered with 2 clusters and 4K splats, level 0+1 rendered with 6 clusters and 12K splats, and finally the whole model rendered with a pixel tolerance of 1 with 1720 clusters and 3435K splats.

view-frustum culling can be applied. Another disadvantage is that it is limited to objects that fit completely into the memory of the graphics card, since the VBOs it uses to draw the models are stored in graphics card memory. Finally it needs a global LOD selection, so even if parts of the model could be rendered at a coarser LOD, the LOD is the same for all parts. Therefore more points than necessary have to be processed.

2.2.4 Layered Point Clouds

Layered point clouds [GM04] use a quite different data structure for hierarchical rendering compared to the two afore-mentioned algorithms. Instead of creating new points which average the contributions of their children in the hierarchy, no new points have to be created for hierarchical rendering. This is a consequence of the data storage used by layered point clouds.

The input is a set of evenly sampled points which are surrounded by a bounding box. Then a subset of these points is extracted by a heuristic method, such that the points in the resulting subset are again evenly distributed. This means that the distance between two points of the subset is close to an average value r.

This average value will then be used in the rendering stage (see below). The remaining points are divided in two groups at the midpoint of the longest axis of the bounding box. The two groups are then recursively subdivided, until only a few points < M are left, in which case a leaf node is produced. This means that 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. Figure 2.10 shows how different point clouds contribute to the appearance of the final model. For each level in the hierarchy, the mean distance r between two neighbouring points is known, so a minimum distance between points for the projected point clouds can be given as a matching criterion. If the minimum distance is reached, then the traversal of the hierarchy stops and the model should be completed. The minimum distance also serves as the splat size that will be rendered. As a second memory saving method, the attributes of one sample are quantized, delta encoded and then compressed with the free available LZO compression library. The position is quantized such that the quantization error is inferior to half of the input sampling distance. This results in 13 to 15 bit for each of the X, Y, and Z axis. The normal is quantized to 16 bits. Delta encoding is then applied, which means that for two given values, not both values will be saved, but only the value of the first one, and then the difference to the next one is saved. This can be done for several values, and it is done as a sort of preprocessing for the final step, the entropy encoding with the LZO algorithm. This results in a sample size of 40 bits or less, which is quite good, but no color is saved, compared to the QSplat algorithm. The entropy decoding can be done in real-time with the LZO algorithm.

The hierarchical structure is split in two parts, namely an index tree and a cloud repository. The index tree is used for traversing the structure. One node of the index tree references its associated point cloud with a 32 bit index. A node contains information that is needed for traversal, that is a bounding sphere, the sample spacing, a cone of normals, and the index of the two children. The index tree does not need very much memory, since each node represents M points. M is user defined and can range from 1 to several millions, but this is dependent on the memory of the graphics hardware (see below). The compressed point clouds are stored in a repository, which is accessed by the rendering algorithm through a data access layer. If a point cloud is needed for display, the access layer will fetch it, and decompress it. Since the access layer masks the location of the repository, it can also be on a remote site. Through the use of the data access layer, models can be rendered that do not fit as a whole in the memory of the rendering client. Another benefit of this data structure is that it supports a representation of point clouds that is amenable to hardware acceleration. A disadvantage is that the build up process needs uniformly sampled point clouds as input. So actually only postprocessed models can be used with this algorithm. The visual quality of the

rendered models cannot compete with high-quality rendering algorithms, as only non anti-aliased point primitives are used for rendering.

2.2.5 XSplat

In XSplat [PSL05] Pajarola et al. took the idea of the SPT algorithm [DVS03] one step further, and developed a data structure that allows points to be stored not only in graphics memory, but also in main memory and virtual memory. Therefore they use a block-based sequentialized multiresolution hierarchy. The input is a level-of-detail (LOD) hierarchy H, where the original points are at the leaf cells, and the interior nodes average the information of their children. In this hierarchy each node represents a surfel and stores several attributes, i.e., the position $\boldsymbol{p}$, color $\vec{c}$, normal $\vec{n}$, and bounding sphere radius r. For each node an $rmin$ and an $rmax$ value are calculated, which are the minimum and maximum distance to the viewpoint for which the splat will be used. This is done similar to [DVS03], but Pajarola et al. use a simpler screen-space error metric which only takes into account the projected size of the bounding sphere. This means, that with XSplat it is not possible to adjust the point size according to the local curvature of the object or to changes in the geometry, like surface edges. In the sequentialized hierarchy the $rmin$ and $rmax$ values are used as a substitute to the hierarchical traversal. This is possible because the two distances can be used to decide if a point should be rendered without the knowledge if the parent cell or the children cells will be rendered as well. For each node also a layer index l is calculated, which is the length of the longest path from that node to a leaf node in the subtree below it. The length is expressed in the number of nodes the path consists of. And finally a z-index is calculated for each node, which is a linear index on the leaf nodes that can be generated by a proper traversal of the hierarchy H. An inner node gets the z-index that is the smallest of one of its children. In Figure 2.11 a tree is shown where the layer index and the z-index are already generated, and each node is assigned an index-pair (l_i, z_i).

After all nodes in the hierarchy have the necessary indices, the hierarchy is sequentialized. The sequentialized hierarchy S is then sorted according to the index pairs in lexicographical order, with decreasing layer index and increasing z-index. Figure 2.12 shows the sequentialized and ordered hierarchy of Figure 2.11. The sorting according to the previously found indices causes nodes which are likely to be rendered at the same LOD to be close together in the hierarchy S.

This hierarchy S could be used for rendering models that fit in the memory of the graphics cards. To allow the rendering of larger models, the hierarchy is subdivided into blocks, as depicted in Figure 2.13. The number of blocks is an order of magnitude smaller [PSL05] than the number of points within the hierarchy, and therefore the list of blocks B can be managed in main memory. Each block in B

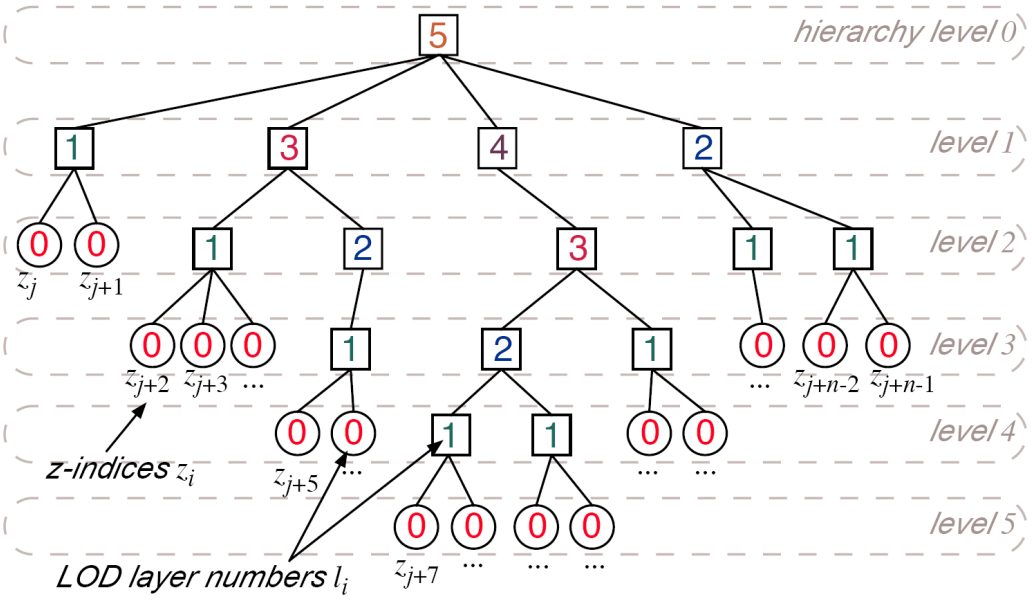

Fig. 2.11: Hierarchical layer-based LOD classification for the nodes. The leaf nodes are linear ordered in z-index.

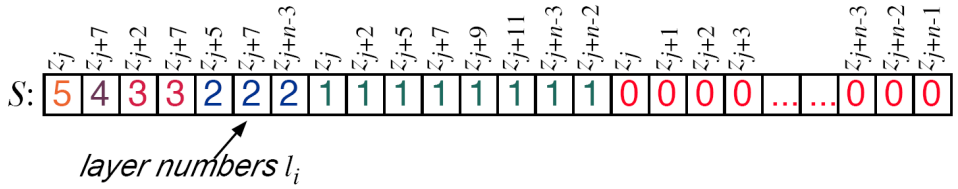

Fig. 2.12: Final ordering of nodes in the sequentialized hierarchy S.

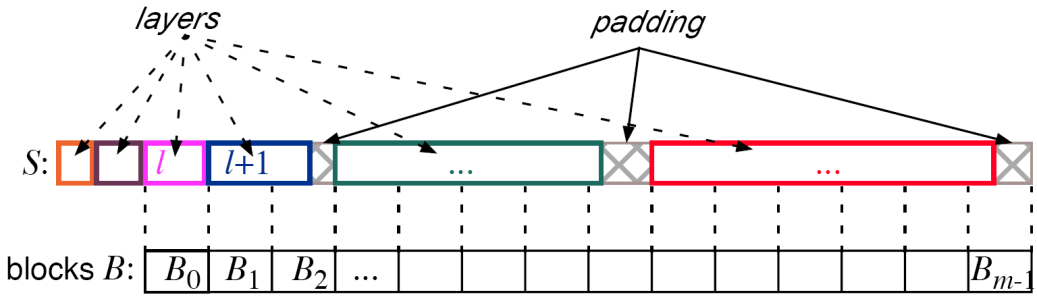

Fig. 2.13: The hierarchy S is subdivided into blocks. Empty spaces in the blocks are padded with NULL points.

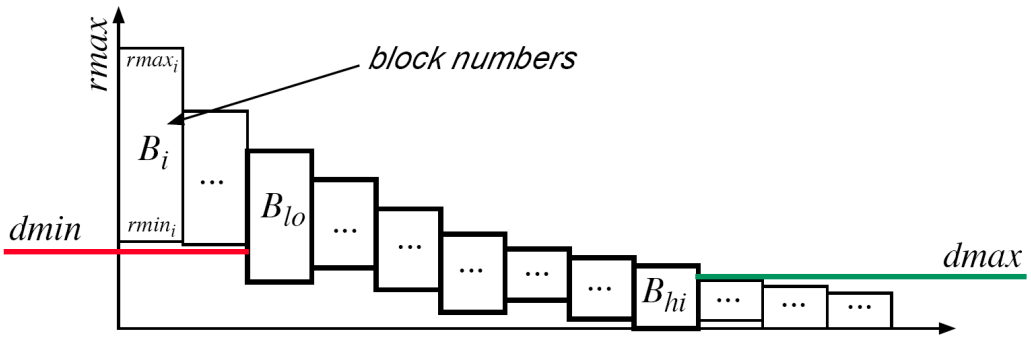

Fig. 2.14: Ordering of blocks in B with respect to the block's rmax values and selection of range.

averages the attributes of the nodes contained within. To enable rendering from the sequentialized representation, it also stores values $rmin$ and $rmax$, which are calculated by $rmin = \min(rmin_i)$ and $rmax = \max(rmax_i)$ from the nodes i contained in the block. The list of blocks B is then ordered by decreasing $rmax$ which can be seen in Figure 2.14.

The rendering in the XSplat algorithm is now as follows. For the current viewpoint v and a user-defined screen-space error tolerance ϵ, a block B_i is only rendered, if the blocks $rmin_i < dmin$ and $rmax_i > dmax$, where $dmin = \sqrt{\epsilon} \times (|\vec{p}_0 - \vec{v}| + r_0)$ and $dmax = \sqrt{\epsilon} \times (|\vec{p}_0 - \vec{v}| - r_0)$. The values p_0 and r_0 are the position and bounding sphere radius of the block with the largest bounding sphere in B. An example is shown in Figure 2.14, where only some blocks of the hierarchy are selected for rendering. The blocks can be cached in the memory of the graphics card to increase the rendering performance. The arrays of S and B are accessed with memory mapped files. The blocks can be swapped in and out of graphics card memory, depending on the coarse LOD block selection for the current viewpoint. The fine-grain LOD selection of single points is accomplished as described in [DVS03].

The XSplat system has the advantage that large models which do not fit into the main memory of the computer can be rendered. For rendering the points can be cached on the graphics card, so hardware-accelerated rendering is possible. Nevertheless Pajarola et al. report only 1.2 FPS for rendering a model consisting of 14M points, which includes the out-of-core processing. The disadvantage of the XSplat system are the requirements of the proposed data structure. The memory requirements for models encoded with the XSplat algorithm are similar to models encoded with the SPT algorithm, and both systems use additionally created points to represent the inner nodes of the original point hierarchy. These additionally created points prevent the systems to reach the maximum number of rendered points per second that a graphics card is capable of, because some inner points

will always be culled by a vertex shader, but they have to be processed as well.

2.3 Summary

From the original idea of using points as rendering primitives [LW85], the further development of point-rendering algorithms can be divided in two areas. In one area the main focus is on rendering the point-based models with high visual quality. In the other area the main focus is on rendering as many points as possible within a given time period. High-quality rendering needs more information per point and uses several rendering passes to create images, therefore the number of vertices per second is not as high as possible. Algorithms developed for the fast rendering of points usually set the visual quality aside in order to be able to render more points per second, which is advantageous for interactive applications. But none of the algorithms is designed to get by with only minimal information available at the points, i.e., the position without a normal associated to it. With minimal information per point the visual quality will not be as good as of any of the aforementioned algorithms, but on the other hand all point clouds can be rendered, even if only a position per point is available.

Chapter 3

Memory Optimized Sequential Point Trees

The main contribution of this thesis is a new algorithm [WS06] which renders point-based models as fast as possible. The vertices per second (VPS) that the algorithm can render are close to the theoretical limit of the graphics card (see also Chapter 6). The data structures that were developed for the algorithm are used for two different purposes. The Memory Optimized Sequential Point Trees (MOSPTs) described in this chapter are used for holding the actual points the model consists of, whereas the Nested Octree described in Chapter 4 is used to organize the MOSPTs in a way that allows for out-of-core rendering of a model. A smaller point cloud that fits entirely in the memory of the graphics card can be converted to a MOSPT, without using the Nested Octree structure.

Both data structures were developed for rendering point clouds that have only minimal attributes. Such point clouds are typical for the output of range scanners. With only position and sometimes color available at a point sample, the presented algorithm can be used for fast reviewing of the scans coming directly from the scanner without the need for postprocessing.

3.1 Motivation

A MOSPT is a new sequentialized hierarchy which is optimized for usage in the memory of the graphics card. The points and associated colors within the hierarchy are stored in arrays in the memory of the graphics card, and therefore the graphics processing unit (GPU) can access and process them very fast during rendering. A MOSPT *uses only the points of the original point cloud*, and it does not create any additional points for the inner nodes. This is much in the sense of vertex clustering algorithms, as described by Luebke [LE97] and Rossignac [RB93], where one vertex is taken from the input vertices to represent all vertices in a hierarchy node. A MOSPT only needs the positions of points to work properly, and there is no need for normals per point. This can

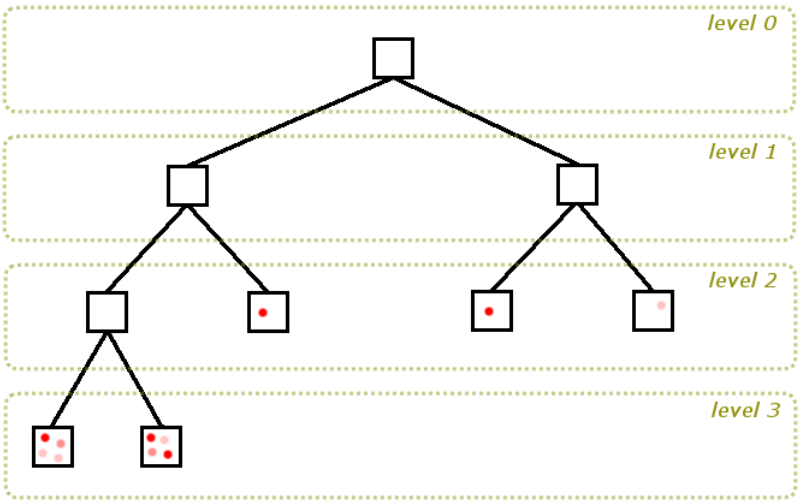

Fig. 3.1: A binary MOSPT after all points have been inserted. Only leaf nodes contain points. The leaf nodes at the maximum recursion level can contain more than one point, if leaf node strategy 3.a is used.

be of importance if the raw data from a range scan is used as the original point cloud. Raw data from range scanners usually cannot provide a normal for each point. Instead the normals would have to be estimated in a postprocess, which is a time-consuming and especially error-prone procedure. Using a MOSPT as data structure, an efficient LOD algorithm can be used which does not need any additionally created points, and this is in contrast to many other point-based LOD algorithms [RL00, PZvBG00, DD04, DVS03], where one level in the hierarchy represents the model at a certain LOD as a whole.

3.2 Build Up

The input for a MOSPT is a set of points $\mathbf{P} = \{P_1, P_2, \ldots, P_i, \ldots, P_n\}$. The only necessary attribute for one point is the coordinate of the point. Other attributes are not necessary, but if a color is available, it can be used as well. It is not necessary to provide a normal for each point, because the rendering algorithm also works without normals. The downside of not having normals is that no advanced lighting or backface culling is possible.

The points are then inserted into an octree one after another. There are three possibilities when a point reaches a leaf cell:

1. If the leaf cell is empty, the point is stored there.

2. If there is already a point in the leaf cell, then the leaf is split, and both points are filtered down in the hierarchy. The leaf node so becomes an inner node.

3. If the leaf node is at the maximum recursion level for the hierarchy and already contains a point, then there are two different strategies:

   (a) *Add* the point to the node.

   (b) *Reject* the new point and write it to disc in the proper rejection file, because it does not add any new information.

The leaf node strategy 3.a is used either if only one MOSPT will be built up, or if a Nested Octree will be built up, and the number of points is below a certain threshold (see Chapter 4.2).

After inserting the points in the octree, only leaf nodes will hold points. In a second phase, a bottom-up strategy is used to choose a point for all inner nodes that were created during inserting the points. For this, each leaf node chooses a representative. Different criteria could be used here, and we decided to choose the point which has the least distance to the averaged color of the points in the leaf node. The distance between colors is calculated by a formula which takes into account the perceptual color distance [Com06]. The representative is then pulled up the hierarchy, and put into the parent node. The procedure is repeated recursively, until the root node of the MOSPT gets a point. In Figure 3.2 a MOSPT with 4 levels is shown. Note that not all inner nodes hold points. This is a result of the LOD mechanism used by the MOSPT, where each LOD is assembled from the points of a level in the hierarchy together with all points that reside in levels above the chosen level. Each additionally rendered level refines the geometry of the levels above it.

Figure 3.3 shows how the different LOD levels look when using the MOSPT of Figure 3.2. At LOD level 0 the hole MOSPT is represented by only one point. At LOD level 1 the one point of LOD level 0 is used to represent the left part of the MOSPT and the only one point stored at level 1 in the hierarchy (see Figure 3.2) is used for the right part. At level 2 already 4 points are needed to represent the whole MOSPT, but again the points from the upper two levels of detail can be used to represent parts of the LOD. At level 3 finally all points from the original hierarchy (see Figure 3.1) are used. At any LOD only the points of the original hierarchy are used.

The LOD algorithm of the MOSPT relies on the fact that *all points that are within the bounding box of a node in the hierarchy are projected to the same pixel on screen if the node appears as large as or smaller than a pixel on screen*. Utilizing this property, it does not matter which point to choose, e.g., for the root

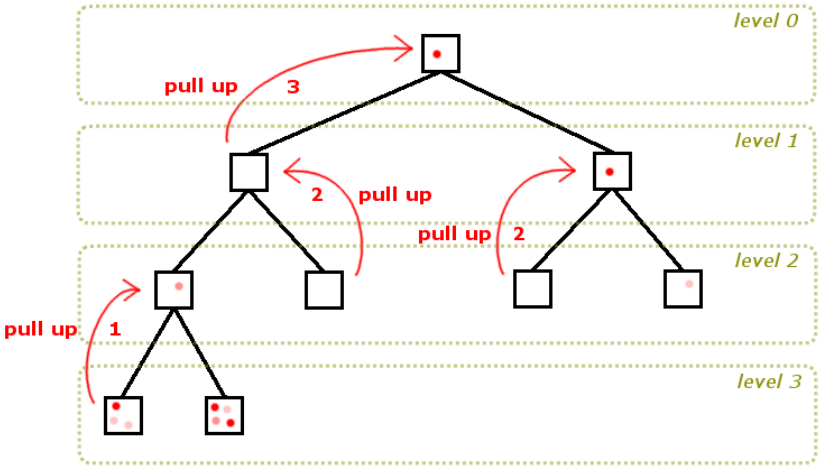

Fig. 3.2: In the second phase a bottom-up algorithm is used that pulls up one representative of the children to the parent nodes. The numbers beside the arrows indicate the iteration step at which the pull-up occurs.

node of the MOSPT. Any point in the hierarchy can be chosen, as they would all be projected to that same pixel.

This is convenient for the build up of the LOD hierarchy, but it also means that aliasing will occur at the higher levels of detail, due to undersampling of the point cloud. For better results the color contributions of the points should be filtered some way to avoid aliasing.

3.3 LOD selection

The points which represent a level of detail in a MOSPT are assembled from all levels above a certain level in the hierarchy. This level is chosen by a screen space error metric. For this, each node in the MOSPT is associated with an error e, and we choose e to be the bounding sphere diameter d of the node. When a node is projected to screen, the error becomes e/r. The value r is the distance from the camera position to the vertex of the object's bounding box that is closest to the camera (see Figure 3.4). This is the same bounding box as for the root node of the MOSPT.

Let ϵ be the error threshold which should not be exceeded by the LOD, e.g., 1 pixel. This means that the projected length of the bounding sphere diameter of the appropriate LOD should not exceed the size of 1 pixel on screen. The test for finding the appropriate LOD can then be written as $e/r < \epsilon$.

The bounding sphere diameters for the levels in the hierarchy are stored in an array, sorted by level number in the hierarchy, beginning with the highest level.

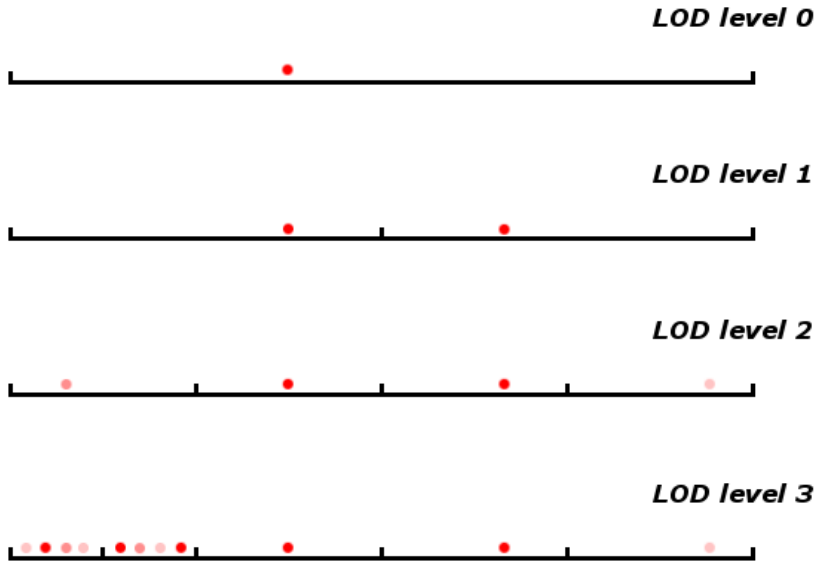

Fig. 3.3: The points of the MOSPT that are used for the different levels of detail.

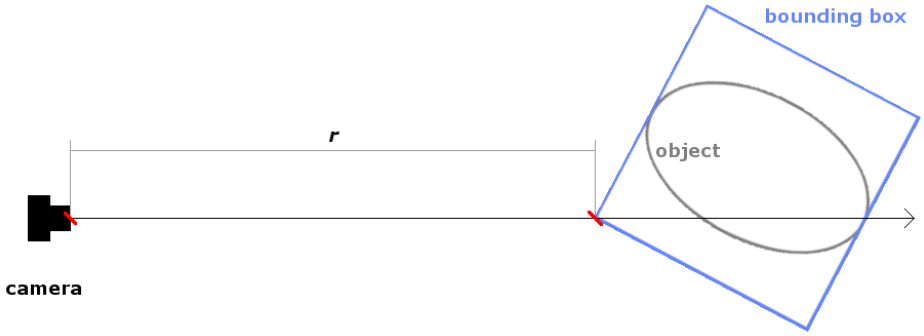

Fig. 3.4: The value of r measures the distance between the camera and the bounding box of the object.

Attribute	size
position p	12 bytes
color c	4 bytes

Tab. 3.1: Attributes of a point in the MOSPT data structure.

For each level the diameter is projected to screen space for the current value of r, and it is calculated how many pixels the diameter takes on screen. The appropriate LOD is then found by simply comparing the projected diameters to the threshold ϵ, beginning at the highest level in the hierarchy. The first level where $e/r < \epsilon$ can then be used for rendering.

3.4 Memory Requirements

A point must store only its position and color. The bounding sphere radii for the different levels are stored outside the MOSPT. Therefore the MOSPT data structure uses only 16 bytes per point (see Table 3.1). Because the bounding sphere radii are not stored with the points, the whole MOSPT can use only one LOD at a time. This means that more points than necessary are rendered. Since a model usually has some extent, not all areas in a model have the same distance to the viewpoint. Some areas could be rendered at a coarser LOD if they are further away from the viewpoint, but when only one LOD can be used, they will be rendered at a LOD that is too fine for their distance to the viewpoint.

3.5 Sequentialization

The hierarchy described so far can be processed by the CPU, but for fast rendering it is preferable to store the geometry on the graphics card. For this the hierarchy has to be transformed into a representation that can be processed by the GPU. Similar to the SPT data structure [DVS03], the hierarchy is sequentialized into an array, with the root node being at the first position. The sequentialization can be achieved with a simple level order traversal of the hierarchy. In the resulting array the points are already sorted by decreasing recursion level, and all points of one level are within a continuous block. In Figure 3.5 the hierarchy of Figure 3.2 is represented in the sequentialized form. The sorting by decreasing recursion level is an optimization that allows the selection of the appropriate LOD by simply rendering a prefix of the array. The optimized speed of the SPT is a result of this selection, not because of the culling of unnecessary vertices in the vertex shader.

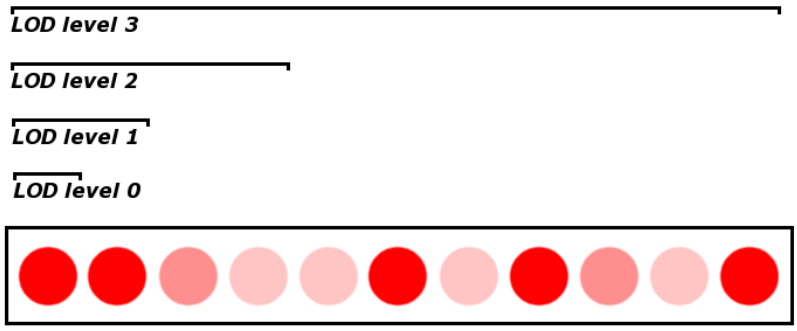

Fig. 3.5: The MOSPT of Figure 3.2 in the sequentialized representation. For each LOD only a prefix of the array has to be rendered. The upper levels of the hierarchy are also used for the lower levels of detail.

3.6 Rendering

The rendering of a MOSPT requires three steps:

1. Find the first hierarchy level at which $e/r < \epsilon$. This is the level which represents the appropriate LOD together with all levels above it in the hierarchy. The viewing distance r is measured at the position of the model that is closest to the camera, e.g., at the closest vertex of the bounding box of the model.

2. Get the number of points i that have to be rendered for this level. This is then the index into the array where the sequentialized hierarchy is stored.

3. Instruct the GPU to render the first i points of the MOSPT.

For rendering no vertex program is needed to process the points of the MOSPT, but it could be useful to implement a simple point size attenuation vertex program for view distance dependent point sizes. There exists the OpenGL command `glPointParameters` which can be used to attenuate the point sizes according to the view distance, but this command slows down rendering speed by some 30%.

3.7 Comparison MOSPT with SPT

The MOSPT data structure is developed starting from the SPT data structure [DVS03]. An SPT is a sequentialized hierarchy (see Chapter 2.2.3), which uses the inner nodes in the hierarchy for a level of detail (LOD) representation of

the model. With this it is possible to render the model with fewer points if the viewpoint of the user is in some distance to the model. The attributes of one inner node of the hierarchy are the averaged values of the attributes of the children of the inner node. This means that an SPT does not only contain the points of the original model in the leaf nodes, but it also contains additionally created points in the inner nodes. These additionally created points have to be processed during rendering, even if some of them are culled by a vertex shader. So the number of processed points always exceeds the number of rendered points when using the SPT data structure. The overhead of the SPT data structure can be described as a combination of a structural overhead O_s and a memory overhead O_m per leaf node. The structural overhead O_s can be computed as [DD04]

$$O_s \sim \frac{1}{\alpha - 1},$$

and is dependent on the average branching factor α of the inner nodes. The branching factor of an inner node refers to the number of children that the inner node has. If a plane is sampled with a range scanner and the points are then sorted into an octree, the branching factor will be approximately 4. Therefore a plane in the SPT data structure has a structural overhead O_s of 33%. A higher branching factor is better, because it results in less inner nodes, whereas a lower branching factor results in more inner nodes. For long range scans the branching factor is around 3, which results in an O_s of 50%.

Each point has to store additional attributes that are needed during rendering the sequentialized hierarchy. These attributes can be thought of as memory overhead, as they are not necessary for rendering the point, when it is not inserted into a hierarchy. Table 3.2 shows the attributes of the points in the sequentialized hierarchy of an SPT. The bounding sphere radius d has to be stored at every point, although the nodes at one level all have the same bounding sphere radius. This is necessary because a node in the sequentialized hierarchy has no information at which level it resides. Alternatively an index to an array of bounding sphere radii could be used as attribute, and the bounding sphere radii are passed as uniform variables to the vertex shader.

The bounding sphere radius together with the split- and merge distance account to an 12 bytes increase in memory for each point, which is 75% of the memory of the original attributes and shall be called the memory overhead O_m of a point. Together with the structural overhead, the total overhead O of the SPT structure can be calculated. Let M be the memory requirements without any overhead, let M_O be the memory requirements including the total overhead, and let N be the number of the original points in the model. Then if a $Point\ with\ Color$ equals 16 bytes, the structural overhead O_s is 50%, and the memory overhead O_m is 75%, so

Attribute	size
position p	12 bytes
color c	4 bytes
bounding sphere radius d	4 bytes
split distance $rmin$	4 bytes
merge distance $rmax$	4 bytes

Tab. 3.2: Attributes of a point in the SPT data structure.

$$M = N \times Point\ with\ Color,$$

$$M_O = (N + N \times O_s) \times (Point\ with\ Color + Point\ with\ Color \times O_m),$$

$$O = \frac{M_O - M}{M} = 162.5\%,$$

which indicates that the memory requirements of an SPT are more than double the memory requirements of the original point cloud! The MOSPT data structure is able to avoid the structural overhead as well as the memory overhead, while still providing an efficient LOD selection. The SPT data structure needs the additionally created points for the inner nodes, to render the model at a coarser level of detail. The MOSPT data structure only uses the points of the original point cloud and can avoid the additionally created points. Therefore it can render the same model with more FPS. In Figure 3.6 the SPT hierarchy is shown that results from the hierarchy created in 3.1. Compared to the MOSPT hierarchy shown in Figure 3.2, the SPT hierarchy has to create additionally points for the inner nodes.

The error metric for a MOSPT is the projected size of the bounding sphere diameter in screen space. The test for the appropriate LOD is to find the level in the hierarchy where $e/r < \epsilon$ with r being the distance of the model to the viewpoint, e being the error associated with the node, and ϵ being the error threshold. This error metric can only be evaluated on the CPU. To be able to evaluate this error metric on the GPU, the test could be rewritten by defining $r_{max} = e/\epsilon$, and then assigning each level in the hierarchy a distance r_{max} at which it should be rendered first. The test for the appropriate LOD would then need to find the last level in the hierarchy where $r > r_{max}$ and render it. Additionally a value r_{min} has to be defined, which is the distance at which a level in the hierarchy is rendered last, given the case that the viewpoint is approaching the model (see also Chapter 2.2.3 and [DVS03]). This conversion is only necessary if the LOD is selected for each point by a vertex shader [DVS03], it is not necessary when using MOSPTs. A MOSPT renders only one LOD, and there is no LOD selection at the points.

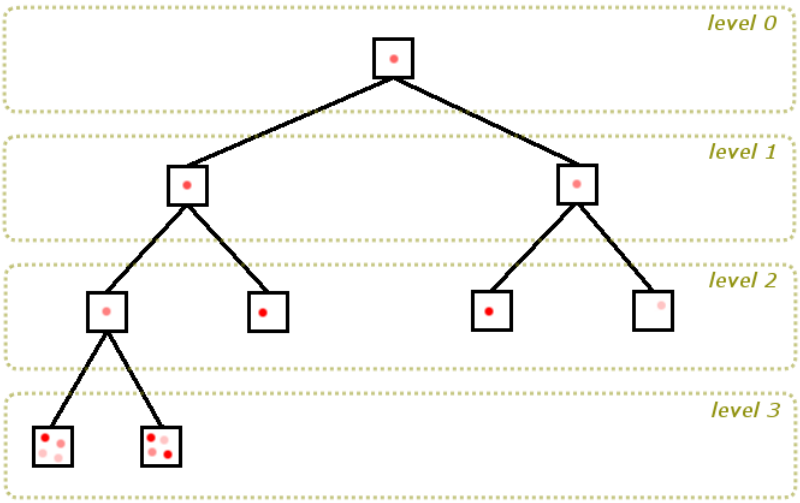

Fig. 3.6: The SPT that results from the hierarchy created in 3.1. The points in the inner nodes average the information of their children.

The SPT actually uses a more sophisticated error metric that tries to estimate the correct splat size depending on the local curvature of the model, as well as on how well a splat covers its children. This is possible because they assume a normal n for each point (see also 2.2.3). The error metric they use results in values for r_{min} and r_{max} that are varying even for nodes at the same hierarchy level.

In an SPT the LOD is really chosen at each node, i.e., regions closer to the viewpoint will be rendered at a finer LOD as regions further away. This is possible, because the bounding sphere radius of each node is stored at the node (see Table 3.2). But there is no advantage to this behaviour, because the number of points that have to be processed remains the same. The nodes that have to be processed by the GPU are selected by the CPU, and for the regions further away from the viewpoint also the points for the finer levels of detail are chosen, even if they are later culled by the vertex program. In Figure 3.7 the left part shows the cut through the hierarchy, where the r is in a fuzzy zone between $\min(r)$ and $\max(r)$, where $\min(r)$ is the viewing distance to the part of the model that is closest to the viewpoint. In contrast, for a MOSPT the LOD is chosen only once, and it is determined for the part of the model that is closest to the viewpoint (see 3.4), so $r = const$ in the right part of Figure 3.7. The model is then rendered using only one LOD.

The rendering of the SPT is also different than the rendering of the MOSPT. Both data structures store their sequentialized hierarchy in an array. Within the sequentialized SPT hierarchy, the points are sorted by the r_{max} value, within the sequentialized MOSPT hierarchy, the points are sorted by the hierarchy level. In

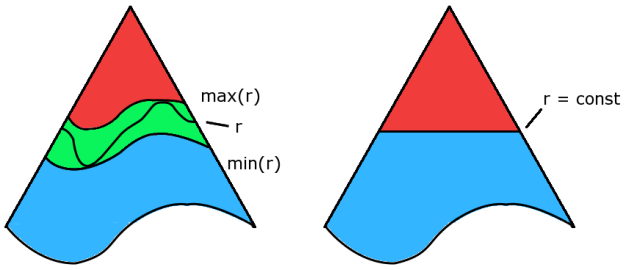

Fig. 3.7: Left: In an SPT, $r = const$ selects different levels of the hierarchy. Furthermore, nodes above and below r need to be culled. Right: In an MOSPT, $r = const$ selects exactly one level in a hierarchy from an MOSPT due to the screen splat error metric.

Figure 3.8 the difference can be seen for the viewpoint at $r = 25$. The MOSPT only selects points from the original point cloud and also renders all of the selected points, while the SPT has to select and process the additionally created points from the upper levels of detail. They are then culled by a vertex program. The MOSPT in contrast does not need a vertex program. Both data structures cannot use view frustum culling, so for viewpoints in the center of the model all points of the data structures have to be rendered. In the case of the SPT, the total number of points is larger than the number of original points, and therefore the rendering of the SPT is even slower than rendering all original points in an unsorted VBO, if the viewpoint is in the center of the model.

Note that the SPT in Figure 3.8 is built up using a different error metric as described by Dachsbacher et al. in [DVS03]. The SPT in Figure 3.8 uses an error metric where only the projected size of the bounding sphere diameter is used to determine the values for r_{min} and r_{max}. The result of this error metric is that all nodes of one hierarchy level have the same values for r_{min} and r_{max}.

The levels of detail for the SPT average the information from the children, therefore the colors in the coarser levels of detail are similar to a low pass filtered image of the model. The MOSPT shows more aliasing, and this is due to the undersampling of the model for the coarser levels of detail (see Figure 6.5).

3.8 Summary

The MOSPT data structure is based on the SPT data structure and tries to retain the speed of the SPT algorithm while reducing its memory requirements. A MOSPT uses only the points of the original model, and only needs to store the position and color of each point, which cuts the memory requirements by more than 50% compared to an SPT. A MOSPT provides an efficient LOD algorithm, and does not require the models to have a normal at each point. It uses a simpler error

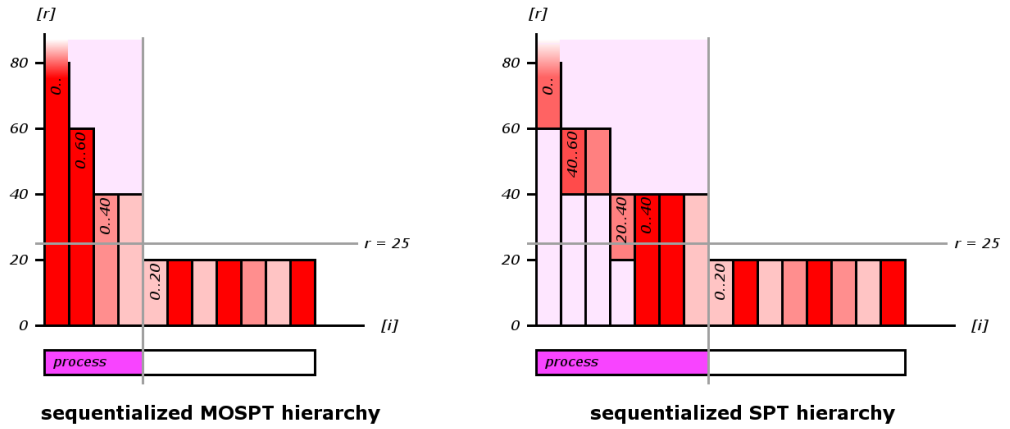

Fig. 3.8: The left side shows the MOSPT resulting from the hierarchy in 3.1, the right side shows the SPT resulting from the same hierarchy. The SPT has to process more points for the camera distance at $r = 25$, but the number of rendered points is the same as with the MOSPT.

metric than the SPT algorithm, so only one LOD can be chosen for the whole model. Finally the hierarchy is sequentialized and can be stored in the graphics card memory, where it can be directly accessed by the GPU for fast rendering. The selection of the appropriate LOD is done on the CPU, which can be done fast. The MOSPT does not allow for view-frustum culling. The size of a MOSPT is limited to the memory of the graphics card. For this, smaller MOSPTs can be embedded into an outer hierarchy, e.g., a Nested Octree (see Chapter 4).

Chapter 4

Nested Octree

Point clouds of large objects can consist of several 100 Million points, and for such huge point clouds special data structures are necessary, since they usually do not fit in the memory of a computer. For rendering such huge models, an efficient out-of-core strategy has to be used, where only the part of the model which is currently visible is loaded into memory. It is also preferable to find a data structure that allows for using the hardware acceleration of current graphics cards. This is not possible if just a simple octree with a single point per node is used. The traversal of the octree does not allow for an appropriate usage of the graphics API. With the hardware-accelerated pipeline, it is preferable to divide a model into blocks that can be stored in the memory of the graphics card, so that many vertices are passed to the GPU in one call. It is not so important that the GPU processes only vertices that are actually rendered. If the selection of the vertices is good enough, processing more points than necessary is faster than exactly selecting only visible points for rendering. This problem is similar to rendering large triangle meshes at different levels of detail. GeoMipMaps introduced by deBoer [dB00] use a quadtree to divide a triangle mesh into blocks, and successively refine the blocks depending on the distance to the viewpoint. Geometry clipmaps introduced by Losasso and Hoppe [LH04] center the levels of detail about the viewer, use nested regular grids for the levels of detail, and allow for compression and synthesis of the terrain. Although these two algorithms do not contain an out-of-core part, they take advantage of the hardware-accelerated rendering pipeline.

For the rendering system presented in this thesis, the Nested Octree data structure is used to divide huge point-based models into smaller parts, which can then be rendered using the hardware-accelerated rendering pipeline.

4.1 Motivation

The Nested Octree is a new data structure, that allows to render models that do not fit entirely in the memory of the graphics card. It consists of an outer hierarchy where at each node a MOSPT is stored. The outer hierarchy is an octree, and a

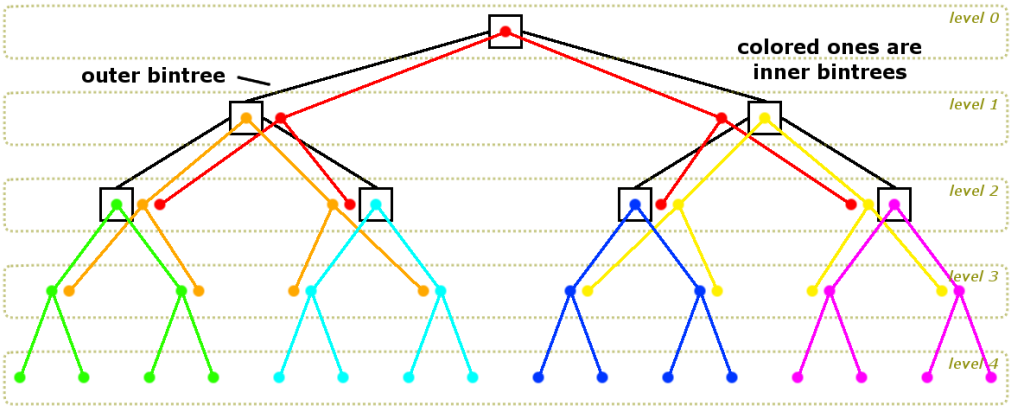

Fig. 4.1: In the one-dimensional case a nested bintree is created. The inner bintrees have a depth of 3. The outer bintree could have more levels as well.

MOSPT is the sequentialized version of an octree (see Chapter 3), which leads to an interesting distribution of points in space, as each MOSPT has cells that overlap with the cells of other MOSPTs. This can be seen in Figure 4.2, where the leaf nodes of the red MOSPT share the same space as the root nodes of the other four MOSPTs. If all nodes would have bounding boxes, than the leftmost leaf node of the red MOSPT, the root node of the green MOSPT, and the leftmost node of the second level of the Nested Octree would all have the same bounding box, because they share the same cell in space.

The MOSPTs in a Nested Octree all have the same number of levels. Let l_{max} be the maximum number of levels per MOSPT, then the maximum number of points that a cell in space can be associated with, is simply l_{max}. This can be seen in Figure 4.1, where $l_{max} = 3$. In the third level, called level 2, there are 3 nodes at each cell in space, and the MOSPT of the Nested Octree's root node ends at this level. A new MOSPT can only be created at the next level, so the maximum number of nodes that share a cell in space cannot increase any more.

In a Nested Octree each point of a cell in space is part of another MOSPT. This can be used for an effective LOD algorithm, because during rendering each added MOSPT of a lower level causes a refinement in the geometry. The Nested Octree also allows for view-frustum culling and out-of-core rendering, which is not possible if the points are stored only in a MOSPT. It would be possible to store all points in an octree, and convert only the lower levels of the octree to SPTs or MOSPTs. The problem with this approach is that the upper levels of the octree also hold a lot of points. They would either have to be view-frustum culled one by one with inappropriate usage of the graphics API, or stored in another data structure. The Nested Octree with MOSPTs included provides a fully integrated solution, from build up to rendering, where the coarsest and finest levels of detail

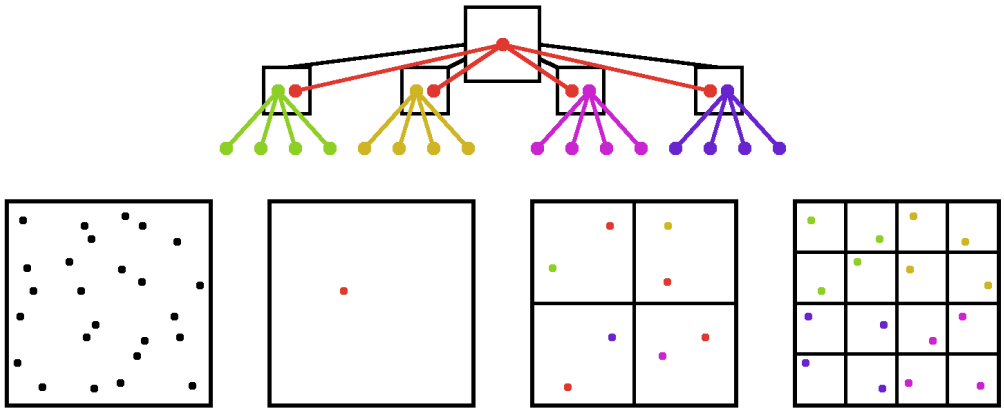

Fig. 4.2: In the two-dimensional case a nested quadtree is created. The inner quadtrees have a depth of 2. In the bottom row the left square shows all available points. The other squares show the points used at the different levels.

are all stored in the same data structure. This is similar to the Layered Point Clouds [GM04], but the build up process for the Nested Octree is simpler and does not rely on a uniform sampling density.

In Figure 4.1 a nested bintree is shown. Each node of the outer bintree holds an inner bintree, which in turn holds the actual points. The cells of bintrees that start at different levels share the same space when the bintrees overlap. The inner bintrees have a depth of 3, whereas the outer bintree could be continued as well. Figure 4.2 shows the two-dimensional case with a nested quadtree. The inner quadtrees have a depth of 2.

4.2 Build Up

The input for a Nested Octree is a set of points, which do not need any attributes, especially they do not need normals. A color value for each point is preferable, or else the shape of the model will not be easily visible. The input point set is first processed to determine the bounding box of the model. The bounding box is then inflated to a cube, which is then also used for the root node of the outer octree. The set of input points may be distributed over several files, e.g., the output files of a range scanner, and can then be composed during build up. The points are sorted into the inner octrees, which later become the MOSPTs. The number of maximum levels l_{max} should not be too small, as this would result in too many files on disc, and l_{max} should not be too large, as this would cause the view frustum culling to become less efficient, because the nodes on the edges of the view frustum will always render all points in it.

Then the root node of the outer octree is created with the bounding box found earlier, and starting with the root node, the following algorithm is performed until all points are sorted into the nested octree.

1. Create an empty MOSPT for the current node.
2. Set the MOSPT leaf-node strategy to *reject* (see Chapter 3.2).
3. Create a rejection file for each child node of the current node in the outer octree, using a unique identifier (see below).
4. Process the input file for this node, and write each rejected point into one of the 8 rejection files.
5. Delete the input file.
6. Set the MOSPT leaf-node strategy to *add*.
7. Check for each rejection file f_i of the current node if the number of points in f_i is below a certain threshold. If f_i is that small, process the rejection file f_i and add all points to the current MOSPT.
8. Pull up the representatives in the MOSPT. After the pull up is finished, the creation of this inner MOSPT is completed (see Chapter 3.2).
9. Write the current MOSPT to disc.
10. Create a new child of the current node in a recursive traversal, and check if a rejection file exists for that child. If yes, make the child node the current node and start another pass of this algorithm.

The unique identifier for each rejection file is created by encoding the path which is needed to reach the node the rejection file is associated with, from the root node of the outer octree. The children of each node in the outer octree are identified with the numbers $1 \ldots 8$, and the root node of the outer octree has the identifier r. Examples for such encoded paths are $r0184$ and $r342$. The identifiers are used as the filenames of the rejection files, so it is easy to check if a rejection file for the current node exists.

The MOSPTs are written to disc in such a format that they can be streamed directly to the graphics card memory and used by the GPU when needed for rendering (see Chapter 4.4).

After all points have been inserted into MOSPTs, the outer octree is written to disc as one file. The outer octree is small enough that it can be held in main memory during the whole build up process. When using an out-of-core build up

algorithm, the number of disc accesses should be kept as low as possible. The described build up algorithm has a theoretical $O(\log\ n)$ time complexity, but this can be alleviated by holding some levels of the inner octrees in memory. If then a point is rejected by a MOSPT, it is possible to try to sort it into the MOSPT of the next hierarchy level. The rejection files are only needed at those levels where the MOSPTs for the following level are not held in memory. After all points have been sorted into the MOSPTs that are held in memory, a rejection file is opened, and new MOSPTs for the necessary levels are created in memory. With the 1GB machine that we used for testing, we were able to hold 3 consecutive levels of inner octrees in memory, which reduces the number of necessary file accesses also by a factor of 3.

4.3 LOD selection

The LOD selection is based on finding the "best" image at any given time. For this we select the point clouds to render in a way similar to Funkhouser and Séquin [FS93], where the geometry at different levels of detail is associated with a benefit and a cost value, and the LODs are selected according to their $Benefit/Cost$ ratio. The benefit is dependent on qualitative and quantitative factors, e.g., the size of the object, or the position of the object in the scene. Then the cost of rendering the geometry at the different levels of detail is evaluated, and the geometry will be rendered at a LOD where the benefit is maximized and the cost does not surpass a user defined threshold.

In our system the cost for all point clouds is assumed to be the same, as we do not use different rendering algorithms at different levels of detail. The benefit of rendering a point cloud is used to sort the point clouds within the traversal queue. The traversal queue is a priority queue, and the first entry of the queue will be rendered first. The calculation of the benefit value is described below.

Let $\vec{p}_{cell}$ be the center of the cell, let $\vec{p}_{viewpoint}$ be the position of the viewpoint, and let $\vec{v}_{cell} = \vec{p}_{cell} - \vec{p}_{viewpoint}$ be the vector of the viewpoint to the center of the cell. The benefit is dependent on the angle α between the view vector $\vec{v}$ and $\vec{v}_{cell}$, and the projected size $\tilde{s}$ of the cell on the screen.

$$\alpha = \arccos(\frac{\vec{v}_{cell} \cdot \vec{v}}{|\vec{v}_{cell}| \times |\vec{v}|}).$$

The projected size $\tilde{s}$ has to be a monotone function. This is important for cells which are partly behind the near plane of the view frustum. Let $z_{nearplane}$ be the z-distance of the near plane to the viewpoint in eye coordinates, let s be the size of the cell on the near plane , and let r be the smallest z-value of a vertex of the cell's bounding box in eye space. Then if a cell holding a point cloud is completely in front of the near plane,

$$\tilde{s} = s/r.$$

If a cell is partly behind the near plane then $\tilde{r} = r - z_{nearplane}$, and

$$\tilde{s} = s \times -\tilde{r}.$$

The benefit of a cell is then calculated as

$$Benefit = \frac{\tilde{s}}{\alpha}.$$

This benefit is used to refine the LOD for cells in the center of the screen first. This is supposed to be the region the user looks at most of the time, so it is better to refine the cells there first. The benefit is calculated and stored at a cell before it is put into the traversal queue. The higher the benefit, the earlier it will be popped from the traversal queue and put into the rendering queue.

It is possible to select the cells in this quite arbitrary order because the Nested Octree algorithm has random access to the point clouds stored on the harddisk. For this we rely on the fast access to files, which is provided by modern filesystems, e.g., the NTFS filesystem uses B-Trees to index the files in large directories [Mic03]. Each point cloud is stored in a separate file (see Section 4.2) and loaded into memory as decided by the LOD selection algorithm, and the files are not memory mapped. Apart from the root node, which is always traversed first, the traversal of the Nested Octree is only based on the LOD selection of the currently available nodes in the traversal queue.

4.4 Rendering

For rendering the Nested Octree, the outer octree is traversed with the help of a *traversal queue*. The outer octree is small and can be kept in memory at all time. Within the traversal queue the nodes are sorted by their benefit (see Chaper 4.3). The traversal always starts at the root node. For the current node that is being processed, the following steps are performed:

1. Check if the node is inside the view frustum. If not, skip the node.

2. Check if the projected size of a cell in the lowest level of the MOSPT that is associated with the node is above a certain threshold. If not, skip the node. This can be done because only the lowest level of a MOSPT refines the geometry of the upper levels (see Figure 4.1). So it is possible to omit the rendering of point clouds whose points would appear too small on screen

and not contribute to the appearance of the model. The threshold can be set by the user, and usually is 1 pixel or 2 pixels. The threshold means the length of one side of the splat that will be rasterized on the screen when rendering a point.

3. Calculate the benefit of the node according to Chapter 4.3.

4. Put the node into the traversal queue.

Within the traversal queue the node will then be ranked according to its priority, and eventually be popped from the traversal queue for further processing. After leaving the traversal queue, the setup calculations for rendering are performed, and the node is finally put into the *rendering queue*, which is a simple FIFO queue:

1. Check if the MOSPT associated with the current outer octree node is available in graphics card memory. If not, request the MOSPT from disk, do not render the node in the current frame, and continue with the next node in the traversal queue.

2. Calculate the level down to which the MOSPT has to be rendered, depending on the view distance r.

3. Put the node into the rendering queue.

The rendering queue is necessary because the calculation of the splat size is dependent on the children of each node. If no children of a node will be rendered, than the splat size calculated for that node will be used. But if children of the node are also rendered, than the splat size of the child that is at the lowest level will be used. In Figure 4.3 an outer octree is shown. The splat size from the leaf nodes is pulled up to the nodes in the higher levels. The root node is then rendered with the splat size from the node, that is furthest down the hierarchy. The splat size is described as the length and width of the splat in pixels. So a splat size of 2 is rasterized with 2×2 pixels.

The splat size of the children will only be used if all children that are visible from the current viewpoint will also be rendered. It can happen that not all children of a node are available, for example if a child is just loaded, or if a child was skipped because it would not contribute to the model. So if this is the case, then the parent node has to render the points with its own splat size.

The user can select the number of points, a *budget*, that will be rendered at most. This can be coordinated with a target frame rate, so that interactive navigation through the model is kept up at any time.

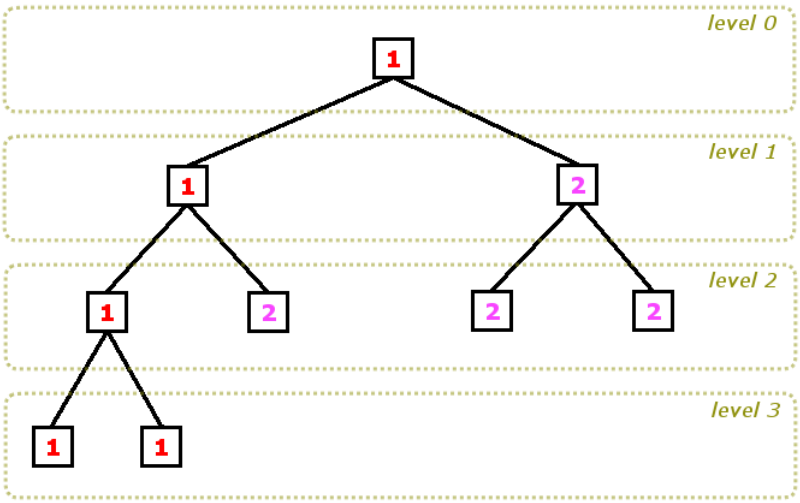

Fig. 4.3: An outer octree, with the splat sizes that will be used for rendering the points of the associated MOSPTs. The splat sizes of the hierarchy levels change with the view distance r.

The requests for the nodes from disk are handled by a separate thread, so that rendering can continue without waiting for the points to be loaded. The points are then streamed directly to the graphics card. On the graphics card a least recently used (LRU) cache is used to determine the point clouds that can be swapped to main memory when a new point cloud is loaded. In main memory another LRU cache is held, and this LRU cache is managed by the second thread which also loads the points from disk. So if a point cloud is requested, it is first searched for in the main memory, and if it is not available, it has to be loaded from disk. The effect of the LRU cache in main memory is that a point cloud which is currently not needed will only be deleted if graphics card memory and main memory are full.

4.5 Occlusion Culling

Instead of simply rendering all nodes that are in the rendering queue, it is possible to add occlusion culling to the algorithm. With occlusion culling, it is possible to avoid rendering nodes that are hidden by nodes in front of them with respect to the current viewpoint. We implemented the occlusion culling algorithm proposed by Gobbetti and Marton [GM04], which uses the hardware occlusion queries that are available on modern graphics cards. The algorithm consists of four steps, outlined below:

1. Find a potentially visible set (PVS) of MOSPTs. This is the usual hierarchy

traversal, but the point clouds are only stored at the rendering queue, and not rendered yet. In the Nested Octree the MOSPTs are stored in the outer octree nodes.

2. Render all outer octree nodes of the PVS that were visible in the previous frame.

3. Issue an occlusion query for all outer octree nodes in the PVS, regardless if they were visible in the last frame or not.

4. Get the result of the occlusion queries. If a visible outer octree node was not already rendered, render it now. All visible outer octree nodes are marked, so they can be identified in the next frame.

The algorithm uses four different queues to store the nodes during the different stages of the algorithm. First the nodes are stored in the traversal queue as described in Chapter 4.3. Then they are put into the rendering queue. When a node is popped from the rendering queue, it is only rendered if it was visible in the last frame. After this all nodes are put into a FIFO *occlusion query start queue*. When a node is popped from this queue, an occlusion query is issued. Finally each node is stored in a FIFO *occlusion query result queue*, from where the nodes are then processed after the query results are available.

For the occlusion queries a node of the outer octree is approximated by the bounding box of that node. For bounding boxes that are completely in front of the viewplane, backface culling is activated. The occlusion culling is effective for large models which use out-of-core rendering, as the results in Chapter 6 show. For small models that completely fit into the graphics card memory, the latency due to the occlusion queries is noticeable. The implemented occlusion culling algorithm is the same as used in the Layered Point Clouds system [GM04], except that with the Nested Octree an additional rendering queue is necessary (see Chapter 4.4). The backface culling is an optimization to the Layered Point Clouds occlusion culling algorithm and reduces the number of rasterized pixels, but the framerate did not increase due to this optimization.

4.6 Comparison Nested Octree with Layered Point Clouds

The Layered Point Clouds algorithm described in [GM04] is a fast point-rendering system. It is similar to the Nested Octree algorithm, as it uses a refinement of points to display different levels of detail. One difference is that the points of the input set for build up should be uniformly distributed for the Layered Point

Clouds, because this is important during rendering also. The build up process relies on the uniform distribution of points to select points in a heuristic manner, and during rendering the splat size is derived from the average distance between points. The Nested Octree does not rely on a uniformly distributed input set of points, neither for build up, nor for rendering. The advantage of one method or the other is dependent on the use case.

As Pajarola et al. mention in [PSL05], the results for the LPC algorithm were acquired with models that completely fit in main memory, as the test machine was equipped with 2GB memory, and all models could be stored in there, because they were compressed. So the reported 40 million splats per second [GM04] were actually measured for a model, that did not use the out-of-core part of the LPC algorithm.

The LPC algorithm orders the point clouds so that point clouds that are close by in space are also close by in the point cloud repository. For access, the file holding the point cloud is memory mapped, and so the coherency pays off. The Nested Octree in contrast relies on the fast access to files in modern file systems, and stores each point cloud in a separate file.

4.7 Comparison Nested Octree with XSplat

The XSplat algorithm described in [PSL05] constructs a block-based sequentialized hierarchy which can be used for out-of-core rendering. The point clouds stored at each block are rendered with the SPT [DVS03] algorithm. This is also the biggest difference to the Nested Octree, as XSplat needs additionally created points to represent the different levels of detail. The memory requirements are similar to the SPT algorithm [PSL05], and therefore the total memory requirements are more than doubled compared to the original point cloud (see also Chapter 3.7). The additional memory is used to render models at a higher quality than it is possible with the Nested Octree, assuming models provide the necessary informations.

Although the XSplat algorithm orders the point clouds differently than the LPC algorithm, the purpose of the ordering is also to store point clouds that are close by in space on disk in close by regions. For access, the file holding the point clouds is also memory mapped, and so the coherency pays off. As already mentioned before, the Nested Octree relies instead on the fast access to files in modern filesystems, and stores each point cloud in a separate file.

4.8 Summary

The Nested Octree is a data structure that consists of an outer hierarchy where at each node an inner hierarchy is stored. The outer hierarchy and the inner hierarchies all share the same space, and overlap each other. With this it is possible to incrementally refine the model for different levels of detail. The build up of the Nested Octree has a time complexity which is below $O(\log\ n)$ and can use the data files that are coming directly from a range scanner. The rendering includes an out-of-core part, and it uses the hardware-accelerated rendering pipeline. The points used for rendering are stored in the memory of the graphics card. Occlusion culling can be used to further speed up the rendering.

Chapter 5

Implementation of the System

In this chapter some implementation specific aspects of the Nested Octree algorithm are discussed. At first the optimal memory layout for vertex attributes is described. We used OpenGL as graphics API, and the results might be specific for OpenGL. The second implementation specific case is to do view-frustum culling in clip space. The third one is the alignment of point clouds to form a large model when the point clouds come from different scan positions.

5.1 Maximizing the Rendering Speed Using VBOs

A vertex buffer object (VBO) is used to store the vertices of a geometric object and their attributes in the memory of the graphics card. As mentioned above, this is advantageous during rendering, because if the information is needed, e.g., the coordinates of a vertex, the GPU has fast access to it. With the information stored in graphics card memory, it does not have to be sent over the system bus (in DirectX the VBOs are called vertex buffers). From the VBO each vertex together with its attributes is processed one by one by a vertex program. The difficulty is, however, to find out how to store the vertices and their attributes so that the graphics card driver and the hardware on the graphics card can access them as fast as possible. For this purpose we wrote a simple C# application where we could change the rendering parameters quickly. Figure 5.1 shows the user interface of the application with the different options.

During the development of the MOSPT (see Chapter 3) data structure, we also implemented the SPT [DVS03] algorithm, although with a simpler error metric. Therefore we had to store at most 3 vertex attributes.

1. The position, in 3 floats, using 4 bytes per float.
2. The color, in 4 bytes.
3. A texture coordinate representing the minimum and maximum index to the recursion level at which a point will be rendered, in 2 short integers, using

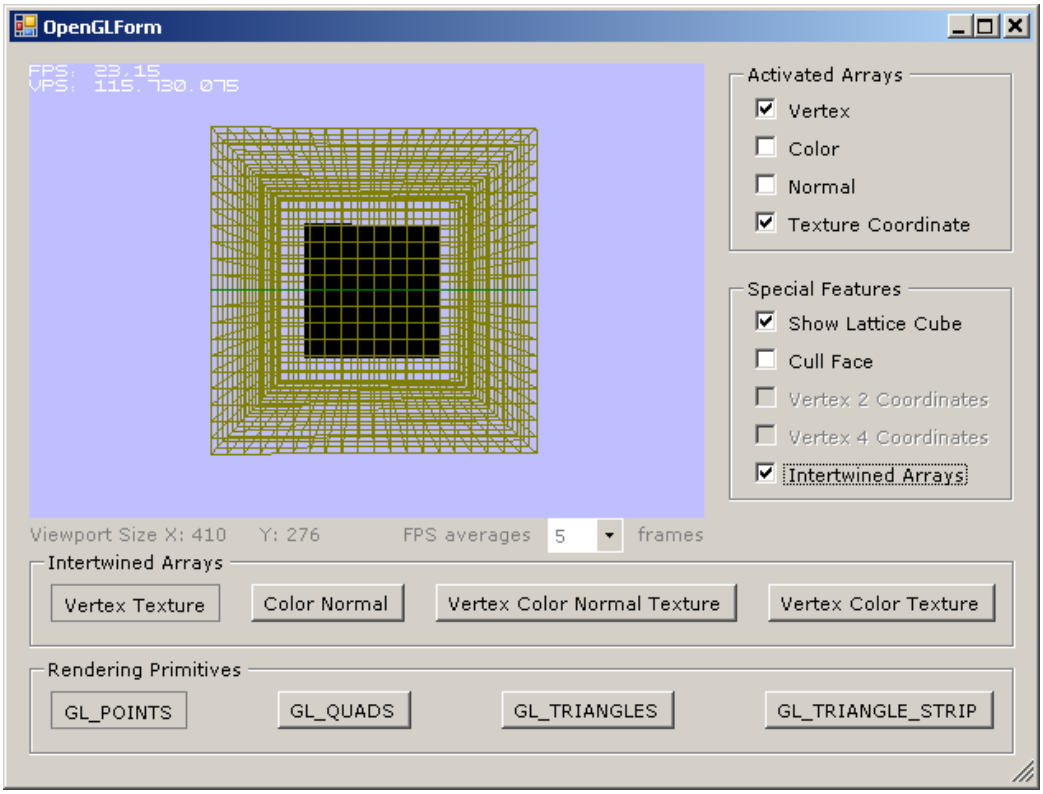

Fig. 5.1: A simple OpenGL application for testing different layouts of vertices in memory.

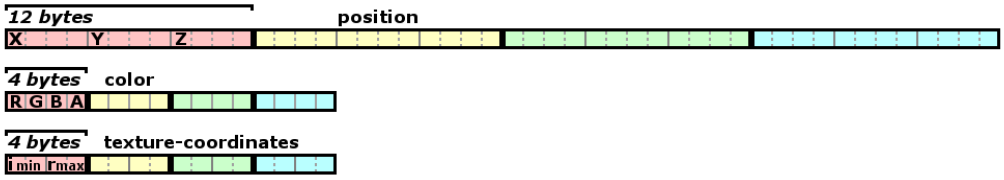

Fig. 5.2: Each vertex attribute is stored in its own array.

Fig. 5.3: All vertex attributes are stored in one array interleaved, at different offsets.

Fig. 5.4: Position and color are stored in one array interleaved, and texture coordinates are stored in their own array.

layout	VPS
3 attribs. in different arrays (Figure 5.2)	103
3 attribs. in one interleaved array (Figure 5.3)	107
2 attribs. in one interleaved array, 1 attrib. in an extra array (Figure 5.4)	116

Tab. 5.1: The results for the different memory layouts.

2 bytes per short integer. Note that the texture coordinate is not used with the MOSPT data structure.

The position and color together need 16 bytes, and if the texture coordinates are also used, then all attributes together need 20 bytes. The different memory layouts are shown in figures 5.2 to 5.4. Independent of the memory layout, there is the possibility to store all arrays in one large VBO at different offsets. This is something different than an interleaved array, where the attributes within an array are stored at different offsets from the corresponding vertex. Using only one VBO for the different arrays has the advantage that the number of VBOs that have to be bound during rendering is reduced. During development we faced the situation that we needed some 10,000 arrays with vertex attributes, and all of them were stored in different VBOs. This finally resulted in a large CPU overhead just for binding the VBOs for rendering.

In the test program 5 Million points are rendered. They are divided into 5 VBOs, each containing 1 Million points. In the user interface the different memory layouts can be selected with buttons. The grid around the 5M points is only rendered for better orientation, because it is possible to move through the world. In the tests we used a GeForce 6800GTO, which has a theoretical maximum of 116.67M processed vertices per second (VPS). This is because of a hardware limitation, as the hardware has a clock speed of 350MHz and requires 3 clock cycles to rasterize a primitive, according to a staff member of NVIDIA. This limitation also means that the maximum number of point primitives per second that can be processed is the same as the maximum number of triangle primitives per second. In Table 5.1 the results from the tests are shown. Clearly the memory layout from Figure 5.4 is the optimum, as points from it can be rendered at maximum speed. The reason for this seems to be that the position and color attributes for one vertex are together 16 bytes long, and that this alignment to 16 bytes can be processed most efficiently by the hardware. In this case even a third attribute can be processed at no cost. Because of these results, in the light of possible future enhancements, we store the color and position of a vertex in the memory layout of Figure 5.4. We do not need the texture coordinates array at the moment, but if it becomes necessary, we could add one more attribute at no processing cost.

5.2 View-Frustum Culling in Clip Space

We evaluated different view-frustum culling algorithms during research. At first we implemented the point-rendering system of the ρ-grids [DD04] algorithm, and used a view-frustum culling algorithm in clip space. With the ρ-grids algorithm, a complete hierarchy is stored in a compressed form, and decoded during rendering. The coordinates of the cell's centers are not stored to save memory. With this algorithm view-frustum culling in clip space is easier to do, because the cell's centers are calculated incrementally in clip space. This method of view-frustum culling worked well also with other algorithms, so we had no reason to exchange it. In the current rendering system this is not a requirement, because the coordinates of the cell's centers do not have to be calculated in clip space. They could be stored at the cells of the Nested Octree as well.

In the rendering pipeline, the clip space is the coordinate system that the vertices reach after the projective transformation, but before the perspective division. For view-frustum culling, an approximation of the cell is needed, and we use the axis-aligned bounding box (AABB) of an outer octree node (see Chapter 4) for this. We do not store the positions of the AABB vertices in the nodes, but instead we store the displacement vectors to the corners of an AABB at each hierarchy level. For reconstructing an AABB on the fly, the center of an outer octree node has to be available. The center of a node also does not have to be stored at the node, only the center of the root node has to be stored. Instead of the centers the displacement vectors are stored at each level. By adding the displacement vectors to the centers of the different levels during rendering, the center of a node can be reconstructed. In Figure 5.5 the right cube shows an octree node with its 8 children. The displacement vectors are used to reach the centers of the children, starting from the center of the parent node.

The sum of the displacement vectors to the centers and to the AABB vertices is calculated in clip coordinates. Only the displacement vectors have to be multiplied with the combined modelview-projection matrix, and then the projected displacement vectors are added. The center of the root node of the outer octree is the only center directly projected to clip space. For this it is multiplied with the combined modelview-projection matrix, a 4x4 matrix,

$$\begin{pmatrix} \tilde{X}_C \\ \tilde{Y}_C \\ \tilde{Z}_C \\ \tilde{W}_C \end{pmatrix} = MVP \times \begin{pmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{pmatrix}$$

and for each level of the outer octree, the displacement vectors are also projected to clip space, by multiplying them with the combined modelview-projection matrix,

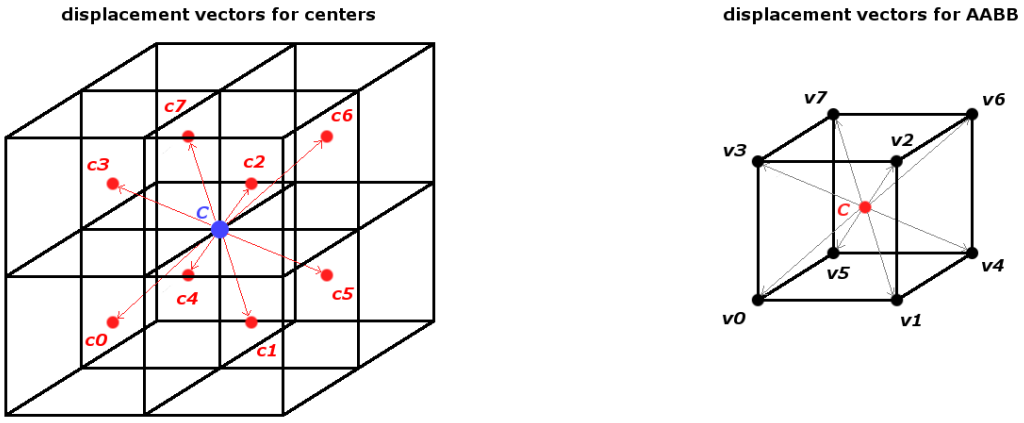

Fig. 5.5: On the left side the displacement vectors from the center of a cell to the centers of its children are depicted. On the right side the displacement vectors from the center of a cell to the vertices of the AABB are depicted.

$$\begin{pmatrix} \tilde{X}_d \\ \tilde{Y}_d \\ \tilde{Z}_d \\ \tilde{W}_d \end{pmatrix} = MVP \times \begin{pmatrix} X_d \\ Y_d \\ Z_d \\ 0 \end{pmatrix}.$$

Note that the displacement vectors have a w-coordinate of 0, because they do not describe a position but a direction in homogeneous coordinates. Then for each level the proper displacement vector is added to the already calculated center of the parent cell, to find the centers of the children in clip space, with

$$\begin{pmatrix} \tilde{X}_c \\ \tilde{Y}_c \\ \tilde{Z}_c \\ \tilde{W}_c \end{pmatrix} = \begin{pmatrix} \tilde{X}_C \\ \tilde{Y}_C \\ \tilde{Z}_C \\ \tilde{W}_C \end{pmatrix} + \begin{pmatrix} \tilde{X}_d \\ \tilde{Y}_d \\ \tilde{Z}_d \\ \tilde{W}_d \end{pmatrix}.$$

The vertices of the AABB of a cell can be calculated similarly, by projecting the displacement vectors from the center of a cell to the vertices of the AABB, and then adding them to the center of the cell in clip coordinates. Having the vertices of the AABBs in clip space, view-frustum culling is done as described by Bloomenthal and Rokne [BR93]. The test is for the maximum and minimum x, y, and w values of the AABB lying either completely outside or inside the view frustum. There are 3 possible outcomes of the test:

1. If they are completely outside or behind the viewpoint, the node is invisible.

2. If they are completely inside, the node and all of its children nodes are visible.

3. If the node is neither completely outside nor inside, the node is rendered, and all children have to be checked against the view frustum as well.

The test requires 10 comparisons at most. View-frustum culling in clip space trades lower memory requirements for extra additions to calculate the AABB vertex positions on the fly.

5.3 Alignment of Point Clouds

When building up a point cloud that consists of several different scan positions, the point clouds usually do not fit together. One problem is that there are erroneous points in the raw data of the scans, which do not belong to the model at all (see Figure 6.7). Another problem is, that the point clouds overlap each other, and the overlapping region should be kept small. If the overlapping regions are not trimmed, then too many points have to be rendered. If the point cloud is scanned indoors, then the lighting of points too far away from the scanner position is wrong. In Figure 6.6 the scans were not trimmed, resulting in black spots all over the model.

Because of this, we try to trim the raw data of the scanner as well as possible, by defining an oriented bounding box (OBB) for each point cloud so that the model only uses points that actually contribute to the appearance of the model. There is no information about the bounding box contained in the raw data from the scanner, therefore it is necessary to measure the extent and the alignment of the OBB directly within the point cloud. The information about the measured OBB is then stored in a text file with each point cloud, and used during the build up of the model.

Figure 5.7 shows an OBB and the possible parameters. The alignment of an OBB can be defined in two different ways, either by giving the orientation of the OBB in angles, or by giving the orientation as normals. They are defined with the keywords “OBB-ANGLES” and “OBB-NORMALS” in the point cloud list, and after the keywords the parameters are given.

The parameters given with OBB-ANGLES are used to set up a rotation matrix. The x-value is used to find the rotation matrix for the x-axis, the y- and z-value are used to find the rotation matrices for the y- and the z-axis respectively. These matrices are then multiplied, and first the rotation matrix for the x-axis is multiplied with the rotation matrix for the y-axis, then the result is multiplied with the rotation matrix for the z-axis. When using this method, the order of rotations has to be respected.

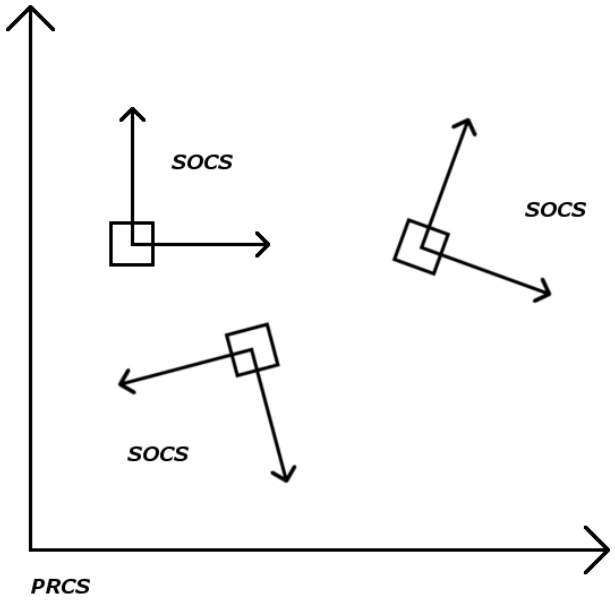

Fig. 5.6: At each scanner position the SOCS coordinate system is in a different orientation relative to the PRCS coordinate system.

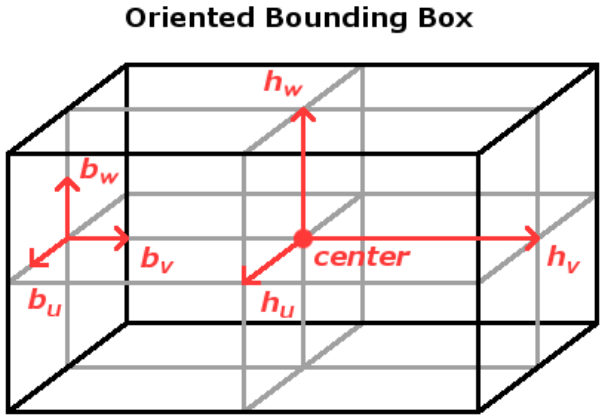

Fig. 5.7: An oriented bounding box is used to cut out the useful part of a raw point cloud. The OBB is defined by b_u, b_v, and b_w, which are the normalized vectors for the local coordinate system of the OBB. Further by h_u, hv, and h_w, which are the length of the half-vectors of the OBB in direction of the base-vectors. And also by the center of the OBB, relative to a superior coordinate system.

OBB-ANGLES parameter	Data
Center of the OBB	3 floats
Half-length for side u	1 float
Half-length for side v	1 float
Half-length for side w	1 float
Angle to axis x of SOCS (PRCS)	1 float
Angle to axis y of SOCS (PRCS)	1 float
Angle to axis z of SOCS (PRCS)	1 float

Tab. 5.2: The center and the angles a, b, and c are all relative to SOCS (PRCS) coordinate system (e.g., the center / angle as seen from the SOCS (PRCS) coordinate system).

OBB-NORMALS parameter	Data
Center of the OBB	3 floats
Half-length for side u	1 float
Half-length for side v	1 float
Half-length for side w	1 float
Orientation of side u	3 floats
Orientation of side v	3 floats
Orientation of side w	3 floats

Tab. 5.3: The center and the normals u, v, and w are all relative to SOCS (PRCS) coordinate system (e.g., the center / orientation as seen from the SOCS (PRCS) coordinate system).

The parameters given with OBB-NORMALS are also used to set up a rotation matrix. With the given vectors an orthonormal basis can be defined, which can then be used as a rotation matrix. The vectors have to be noncoplanar, i.e., when they are written as columns in a 3x3 matrix, their determinant must not be 0.

Besides the orientation of the OBB, the coordinate system in which the OBB is defined has to be given as well. The possible values are “SOCS” for “Scanners Own Coordinate System” and “PRCS” for “PRoject Coordinate System”. SOCS is the coordinate system in which the scanner delivers the raw data. PRCS is a coordinate system which is defined by the user, which is for example an already existing coordinate system at the scan site, e.g., a facility coordinate system. The connection between the PRCS and the SOCS coordinate system is depicted in Figure 5.6. At each scanner position the SOCS coordinate system is in a different relative position to the PRCS coordinate system.

During build up the bounding boxes are applied to the points of a scan. If a point is within its bounding box, then the coordinates of the point are transformed to the PRCS coordinate system.

5.4 Summary

Implementation-specific issues can be dealt with in various ways. The presented solutions work for our implementation, and the memory layout for the fast rendering of VBOs even seems to be the optimal solution. To find the OBBs for all point clouds is a quite time-consuming work, and it would be certainly easier to have a program where the alignment of the OBBs could be done visually.

Chapter 6

Results

The results are divided in two parts. First the Nested Octree and the MOSPT algorithms are compared to the SPT algorithm, with a point cloud that completely fits in the memory of the graphics card, and second the performance of the Nested Octree with a huge point cloud is tested.

6.1 The Test System

For testing we used a Dell Pentium4 3.2GHz computer with hyperthreading enabled, which had a NVIDIA GeForce 6800GTO with 256MB installed. The GeForce has 5 vertex shaders, and a theoretical limit of 116,67 million transformed primitives per second. This is because of a hardware limitation, as the hardware has a clock speed of 350MHz and requires 3 clock cycles to rasterize a primitive, according to a staff member of NVIDIA. In the computer 2 Western Digital Raptor drives were installed as RAID 0 array, each of the drives spinning at 10.000 RPM. The OpenGL viewport has a size of 640x640 pixels, unless otherwise noted.

6.2 Rendering a Small Point Cloud

At first we implemented the SPT algorithm with a simpler error metric than described in [DVS03], because we wanted to have a benchmark for the rendering speed of our new algorithm. For the error metric we only used the projected size of the cell on the screen, due to the lack of normals for our model. We also used a second benchmark, which was a point cloud rendered as a VBO. Both benchmarks do not use view-frustum culling. The model is a range scan from the Stephansdom scanning project, consisting of 6,609,305 points. It is not resampled and fits completely into the memory of the graphics card. In table 6.1 the memory requirements on the graphics card for the different algorithms are shown. Note that the SPT requires less memory than calculated in Capter 3.7, which is due to the simpler error metric. With the simpler error metric it is not necessary to store a

Algorithm	Memory Requirements on the Graphics Card in Bytes
VBO	105,748,880
SPT	200,414,120
MOSPT	105,748,880
Nested Octree	105,748,880

Tab. 6.1: The memory requirements for the different algorithms.

bounding volume at each node. The Nested Octree stores only the MOSPTs on the graphics card, so the memory requirements are the same as for the MOSPT and VBO algorithms. The outer octree of the Nested Octree needs about 500KB in main memory.

We compared the Nested Octree and the MOSPT to the two benchmarks, and chose three different locations for the viewpoint to document the behavior of the algorithms in different situations. In Figure 6.1 the three viewpoint positions relative to the model are drawn in the picture, together with the viewing direction. Figure 6.2 shows the model as seen from the different viewing positions. Note that the visual quality of all algorithms is essentially the same. They all render the model as seen in Figure 6.2. The only difference is that the Nested Octree requests the parts of the point cloud one by one, whereas the VBO, SPT, and MOSPT algorithms store all points on the graphics card before rendering. All algorithms used a constant point size of 1 for the tests. The difference in the LOD algorithms is shown in figures 6.3 to 6.5. The VBO algorithm always renders all points, so it is the benchmark for the visual quality. The difference between the Nested Octree and the VBO rendered image is a bit larger than the difference between the SPT and VBO rendered image, as can be seen in the difference images. The best quality is achieved with the MOSPT algorithm, as can be seen in the difference image in Figure 6.5.

The Nested Octree uses undersampling to represent the different levels of detail. As can be seen in Table 6.3, the Nested Octree is able to choose a coarser LOD when the viewpoint is at the border of the model than the other algorithms, so it renders the model with fewer points. This also means that not all points of the original model are available, and the appearance is only approximated. Therefore the visual quality is a bit worse compared to the other algorithms.

The SPT uses the averaged color of the original points to represent different levels of detail. The SPT can also choose the LOD within the model. Points that are further away are rendered at a coarser LOD. The points in the middle of the screen are furthest away from the viewpoint, and exhibit some error because they are rendered at a coarser LOD than the other points in the image. The points not in the middle of the screen are rendered at the finest LOD. This means that original points are rendered, so there is no difference in the VBO rendered image.

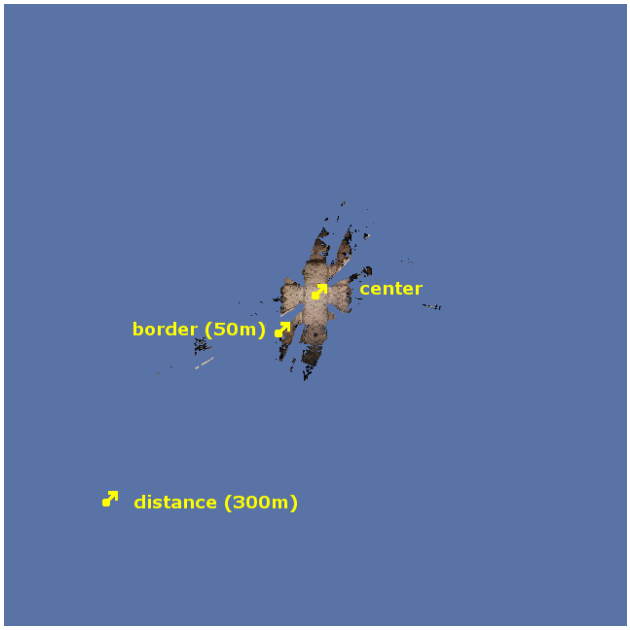

Fig. 6.1: The different viewpositions as seen from above. The model consists of 1 point cloud.

The MOSPT uses the finest LOD for the viewpoint at the border of the model. The whole model is rendered at the same LOD, so all points of the original point cloud are shown. Therefore the errors in the difference image are very small, theoretically there should be no differences at all. But the order in which the points are rasterized is different to the VBO algorithm, so is is possible that the depth buffer is filled in a different order, and if some points have the same distance to the viewpoint, different points may be used to represent the same pixel.

In Table 6.2 the resulting FPS are shown. The model stored in the VBO already gives an interesting result, because the FPS vary, although always the same number of points is processed by the GPU. At the first view position all points are visible and the model is rasterized quite small (see Figure 6.2), so many pixels have to be overdrawn during rendering. This slows down rendering, because due to overdraw many pixels in the depth buffer have then to use a read-modify-write cycle. It is especially slow if a pixel modifies its value several times during rendering a frame. When the viewpoint gets to the center of the model, almost no pixel has to be overdrawn, and therefore the peak FPS is reached at that position. The SPT algorithm shows quite a different behavior. At the viewpoint in the distance the FPS reach their peak. Here the LOD algorithm of the SPT works efficiently. For the viewpoints at the border and at the center, the FPS fall off, and eventually become even lower than the FPS of the VBO algorithm. So an SPT alone shows

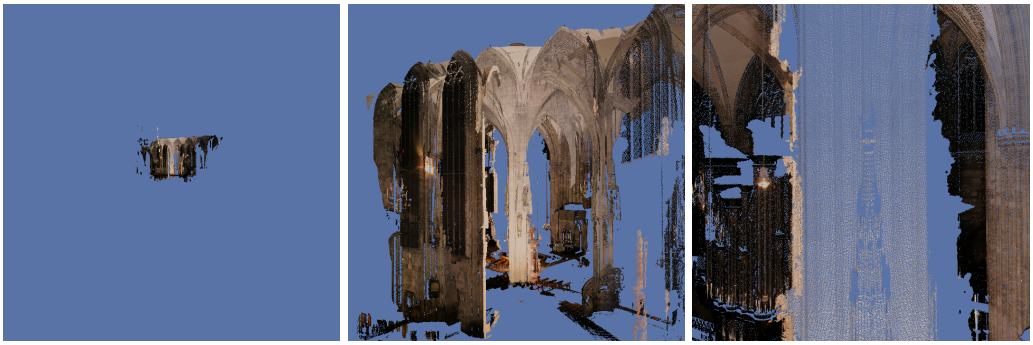

Fig. 6.2: The model as seen from different viewpositions. In the left picture the viewpoint is 300 meters from the center, in the middle picture 50 meters, and in the right picture the viewpoint is at the center of the model.

Algorithm	Distance	Border	Center
VBO	15	17	18
SPT	240	12	12
MOSPT	308	17	18
Nested Octree	722	49	122
Nested Octree with occlusion culling	428	44	100

Tab. 6.2: The Frames per Second for the different algorithms at different viewpoint positions.

deficiencies if the user wants to walk through the model. The MOSPT is the combination of the VBO and SPT algorithm, and is always faster than the SPT and never slower than the VBO. The Nested Octree is the algorithm with the highest FPS at all positions, because it can use view-frustum culling and a LOD selection within the model. It gets even close to real-time performance (where 60 FPS are minimum). The Nested Octree was also tested with occlusion culling enabled, and here the latency due to the occlusion queries is noticeable. The latency is especially high for the viewpoint positions at the distance and at the center.

Table 6.3 charts the number of processed points at the different viewing positions. Here the efficiency of the different LOD algorithms can be seen, for example between the MOSPT and SPT level-of-detail algorithm. The MOSPT uses only original points, whereas the SPT has to process additionally created points in addition, although some of them are culled by the vertex shader. The VBO algorithm always processes all points of the model, and at the border and center positions it achieves even more FPS than the SPT algorithm. Both Nested Octree algorithms use the same number of points to render the model, which indicates that the occlusion culling is not able to cull anything, since the depth complexity is too low for the viewpoints.

Algorithm	Distance	Border	Center
VBO	6,609,305	6,609,305	6,609,305
SPT	456,974	10,021,473	10,021,473
MOSPT	336,869	6,609,305	6,609,305
Nested Octree	122,357	2,335,790	908,295
Nested Octree with occlusion culling	122,357	2,335,790	908,295

Tab. 6.3: The number of rendered points for the different algorithms at different viewpoint positions.

Algorithm	Distance	Border	Center
VBO	101M	108M	116.1M
SPT	109M	116M	116M
MOSPT	104M	112M	116.1M
Nested Octree	88M	114M	111M
Nested Octree with occlusion culling	52M	102M	91M

Tab. 6.4: The vertices per second (VPS) for the different algorithms at different viewpoint positions. The VPS were measured with performance counters, available on the graphics card.

In Table 6.4 the vertices per second (VPS) are charted. The numbers are read from the performance counters on the GPU. Performance counters make it possible to monitor and analyze the behavior of a physical component such as the GPU. Different performance counters exist, like the number of vertices processed each second, or the number of pixels shaded per second. They can be read with a program during rendering. For this we used the NVIDIA Developer Control Panel, and an instrumented driver for the graphics card. This makes it easy to compare the performance of different algorithms.

The read numbers for the vertices per second are only exact within a range of $\pm 1\%$, but they give good indications of how efficiently the GPU is used. The maximum throughput is 116,67 MVPS, which is never achieved in any test. The VBO and the MOSPT algorithms are the fastest of all, with 116,1 MVPS, when the viewpoint is at the center of the model. Here the SPT algorithm shows that it uses the GPU very efficiently with 116 MVPS. The Nested Octree algorithm without occlusion culling shows similar results as the VBO, SPT, and MOSPT algorithms, only for the viewpoint position in the distance the VPS fall off. This is due to the small number of points that are selected by the Nested Octree LOD algorithm at this position. Here the rendering loop on the CPU is the bottle neck. When occlusion culling is enabled, the Nested Octree algorithm performs quite badly, because the latency for waiting on the occlusion query results restricts the achievable VPS.

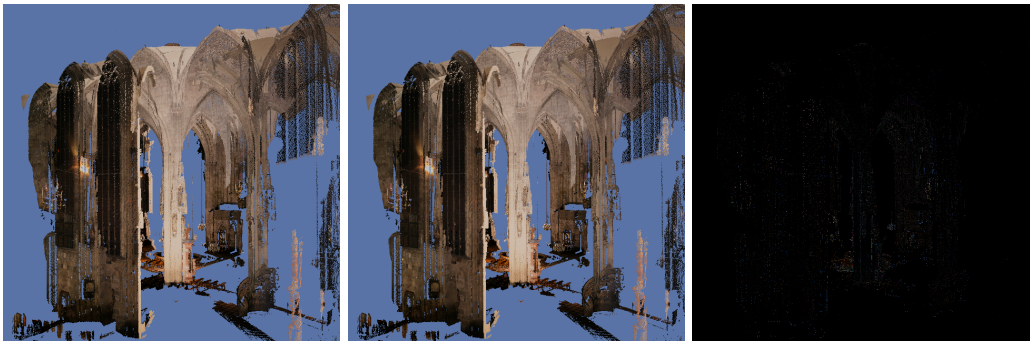

Fig. 6.3: Quality comparison of Nested Octree (left) and VBO (middle) rendering. The VBO represents the original model. The Nested Octree uses an LOD algorithm which is able to select an LOD level for the model, so that the model is not rendered at the finest resolution. There is a difference (right) between the VBO and Nested Octree rendered images.

The Nested Octree algorithm without occlusion culling can be used for small models that completely fit in the graphics card memory. It outperforms the VBO, SPT, and MOSPT algorithms by far at any position that was tested. Only for viewpoint positions that are further away, the VPS fall off compared to the aforementioned algorithms, but this is due to the effective LOD selection of the Nested Octree which renders only few points at this position. When small models are rendered, it is not usefull to enable occlusion culling, because the depth complexity of the model is too low, and so no points can be culled.

6.3 Rendering a Huge Point Cloud

The Nested Octree is a data structure tailored to the requirements of rendering huge point clouds with 100M points and more. The model we used for this was a scan of the Vienna Stephansdom, the largest cathedral in Vienna, and a landmark. The Stephansdom was point sampled during a campaign that lasted a week. Scanning was always performed at night, where no visitors would disturb the work. The range scanner used during the campaign belongs to the TUW-ILScan center of competence, and the campaign was also planned and accomplished by TUW-ILScan. TUW-ILScan is a group of scientists and institutes of the Vienna University of Technology that seeks to improve the Image Laser Scanning (ILScan) technology and also to develop application-oriented implementations for it. The ILScan technology is developed by Riegl LMS, a company which produces range scanners. They are also a cooperation partner of TUW-ILScan. With the ILScan technology it is possible to colorize range scans from photos taken during scanning (see Chapter 1.3.1). This is the technology that was used for scanning the

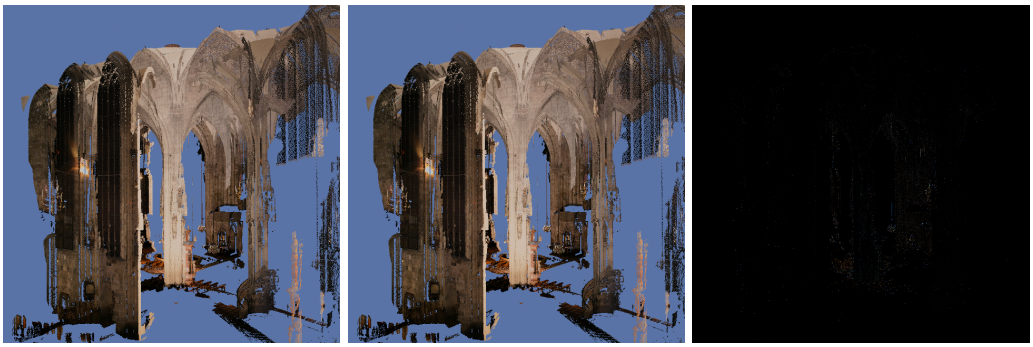

Fig. 6.4: Quality comparison of SPT (left) and VBO (middle) rendering. The VBO represents the original model. The SPT uses an LOD algorithm that is effective in the center of the model. In the other areas, the model is rendered at the finest resolution. There is a difference (right) between the VBO and SPT rendered images.

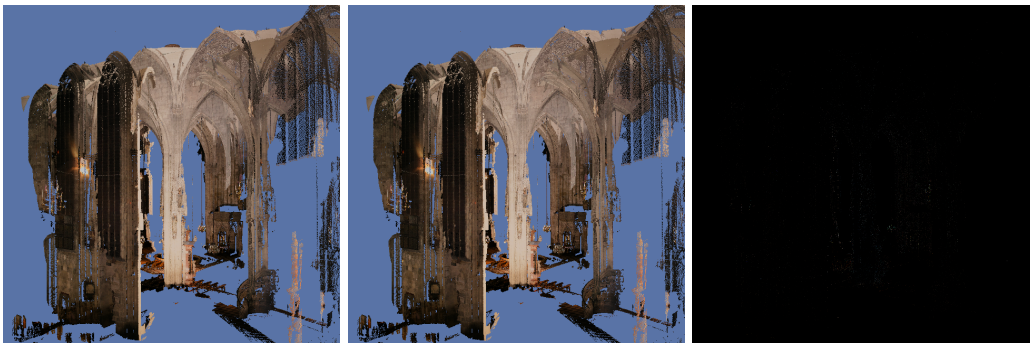

Fig. 6.5: Quality comparison of MOSPT (left) and VBO (middle) rendering. The VBO represents the original model. The MOSPT uses an LOD algorithm, but the whole model is rendered at the finest resolution. There is nearly no difference (right) between the VBO and SPT rendered images.

Fig. 6.6: On the left side the model without trimming the scans. The points in the distance do not receive enough light, and appear very dark or even black. On the right side, the size of the OBB relative to the raw data from a scan as seen from above.

Stephansdom.

For lighting the Stephansdom, spotlights were necessary. This causes an uneven illumination at the different scan positions. The other effect of the working at night is that point samples in some distance received less light than the samples at the scanner position, or none at all. Due to this, the color of the point samples becomes darker the further away they are from the scanner position. This is visible in Figure 6.1, which is the result of one point scan. The scan is actually done for the piling directly above the scanner, so the useable part of the scan measures only 10x10 meters on the floor, and 25 meters in height. The result of not removing enough points can be seen in Figure 6.6, where the black points make the model look ugly.

For sampling the whole cathedral, more than 200 different scanner positions were necessary, but in the model we actually used only 77 scan positions, because this is enough to render a somewhat complete model of the Stephansdom. The other scan positions could be used to refine the geometry. The point clouds from the different scan positions overlap, and so it is necessary to cut out the useful parts of the point scans to achieve a somewhat uniform model when the point clouds of different scanner positions are composed. In the raw data of the scans, there also existed point samples in areas far off the actual model, as can be seen in Figure 6.7. These points are the result of measurement errors, caused by specular surfaces, where the laser from the scanner is reflected. If the reflection recurses over some stations, the error becomes very large and results in a “jet” like structure, as can

Fig. 6.7: Points at weired positions in space. The Stephansdom was scanned from the inside. These erroneous points have to be culled in a postprocess, to get a somewhat clean model. The cleanup process can be automated. The “jet” in the left image should be cut, and the “flying windows” in the right image are undesired as well.

be seen in the left image of Figure 6.7.

To trim the point clouds, an oriented bounding box (OBB) is defined for each point cloud as described in Chapter 5.3. The OBBs of neighboring point clouds should overlap a bit, so that there is no gap between the point clouds. On average, 50% of the points in the scans can be removed. The measurement of the OBB was done for each point cloud individually, as the different point scans differ with respect to position and alignment relative to the scanned case bay. The different relative positions are a result of the interior design of the Stephansdom, and the case bays do not always have the same measurements. So it was necessary to measure all OBBs by hand. The trimming of the point clouds was then performed during build up, so that an intermediate storage of the trimmed point clouds was not necessary. The complete build up of the model, which finally consisted of 262M points, took about 2 hours and 30 minutes, which includes the initial step of finding a bounding box for the model. The model needs 4.2GB on disk for the 37,232 MOSPTs, and 14MB for the outer octree.

In Figure 6.8 the complete model of the Stephansdom is shown. The scans were taken from inside the cathedral, so the arches can be seen as if the roof is missing. Figure 6.9 shows the model from the south-east corner. Figure 6.10 shows the Stephansdom from inside, after moving very fast. The incremental refinement cannot keep up with the motion, and therefore a coarse LOD is rendered. Due to missing points, the LOD is rendered with large splats. The geometry of

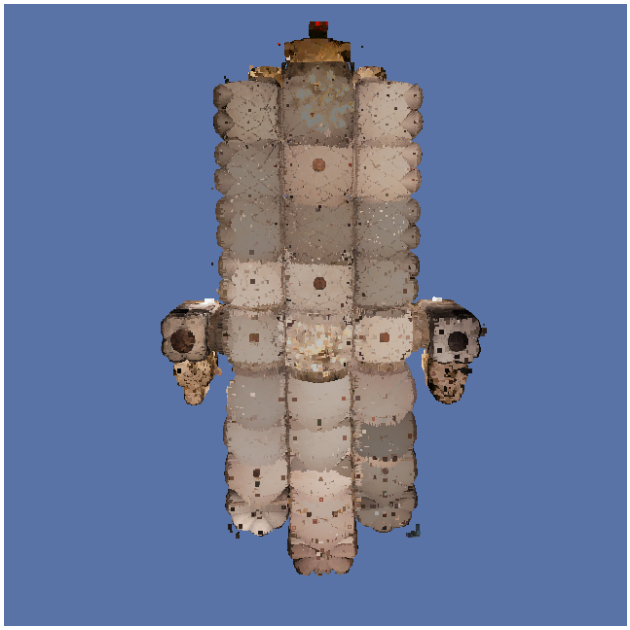

Fig. 6.8: The Stephansdom as seen from above. All 77 scan positions are visible. At the top is the main entrance, and at the bottom the Albertinian choir.

the interior design is nearly unrecognizable.

After 5 seconds the necessary point clouds are streamed in from harddisk to the graphics card memory. Figure 6.11 shows the same position as Figure 6.10 after loading has completed. The minimum size for MOSPT leaf nodes projected to screen is 2 pixels. This resolution is very reasonable for moving quickly through the model, and not waiting too long for the point clouds to load. After waiting another 16 seconds, or 21 seconds overall, the model is refined further, and the minimum size for MOSPT leaf nodes projected to screen is then 1 pixel. This is the highest achievable visual quality, but it takes considerably longer to load.

The reason for the long loading can be seen in figures 6.13 and 6.14. The number of MOSPTs that have to be loaded is way smaller for a projected cell size of 2 pixels than for a projected cell size of 1 pixel. When using a threshold of 2 pixels only 252 MOSPTs have to be loaded, whereas when using a threshold of 1 pixel, 559 MOSPTs have to be loaded. In the figures, each MOSPT is shown with its bounding box. The magenta bounding boxes mean that not all children of that outer octree node are rendered, although they are inside the view frustum. The reason they are not rendered is that the projected size of the child nodes is too small, so they would not contribute to the appearance (see Chapter 4.4). The more saturated the magenta is, the more children are skipped rendering. The number of bounding boxes that are visible is the same, no matter if occlusion culling is

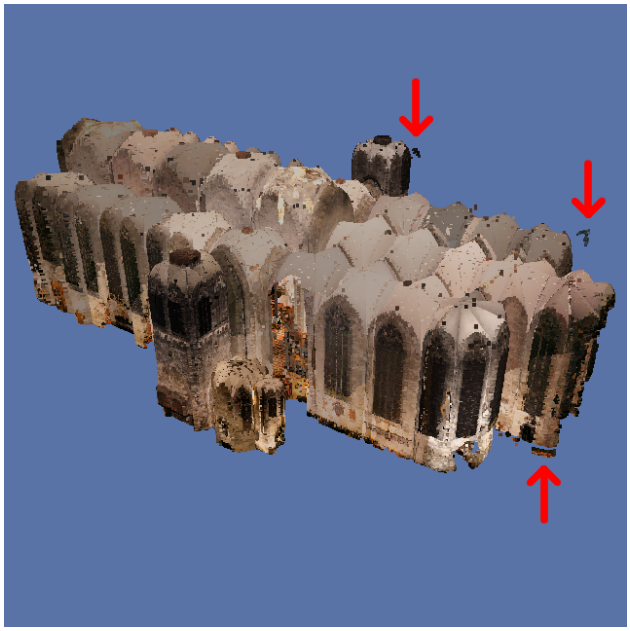

Fig. 6.9: The Stephansdom from outside, looking from south-east to the north-west. Along the right border and below the Albertinian choir still some erroneous points can be seen, which were not cut by OBBs.

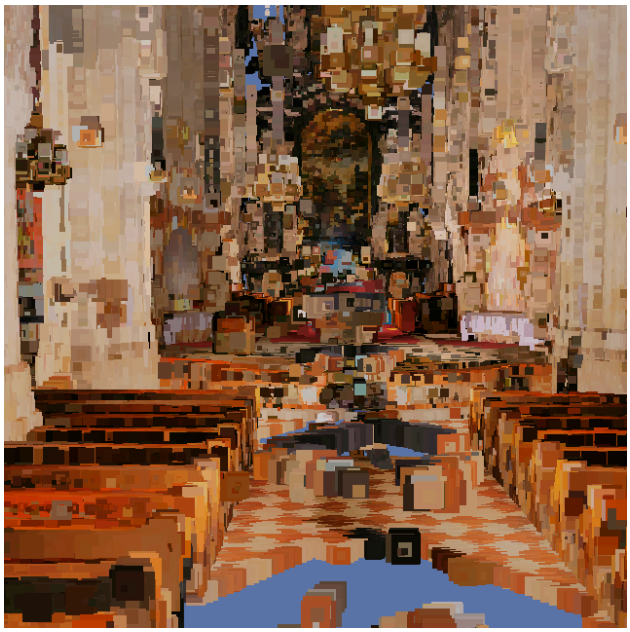

Fig. 6.10: The inside of the Stephansdom. After the user has moved very fast through the model, the interior design is rendered very blocky.

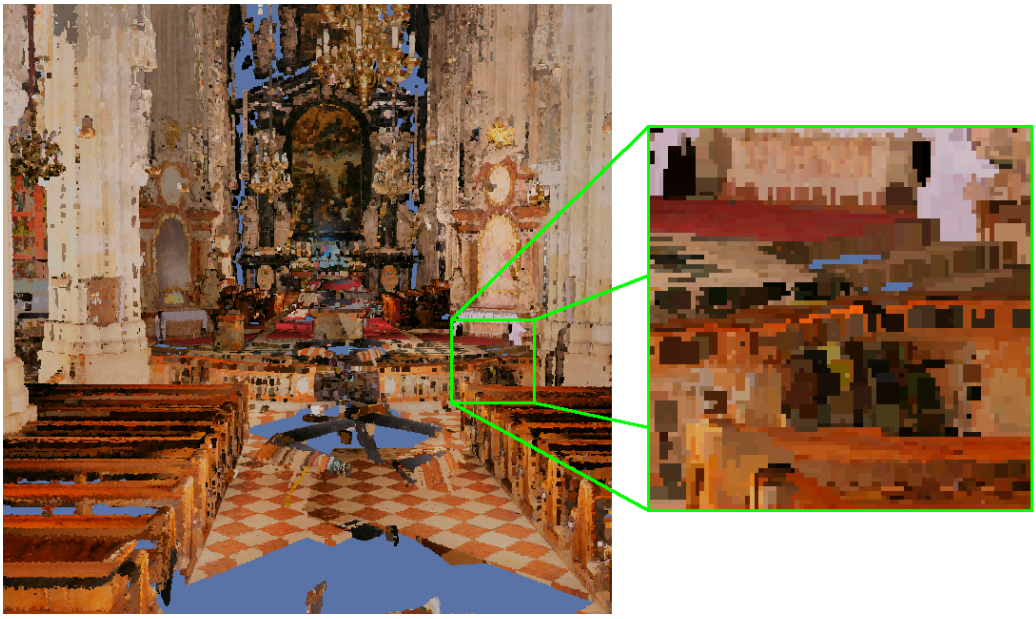

Fig. 6.11: The inside of the Stephansdom rendered with a minimum projected size for the MOSPT leaf nodes of 2 pixels. This is a very reasonable compromise between rendering speed and visual quality.

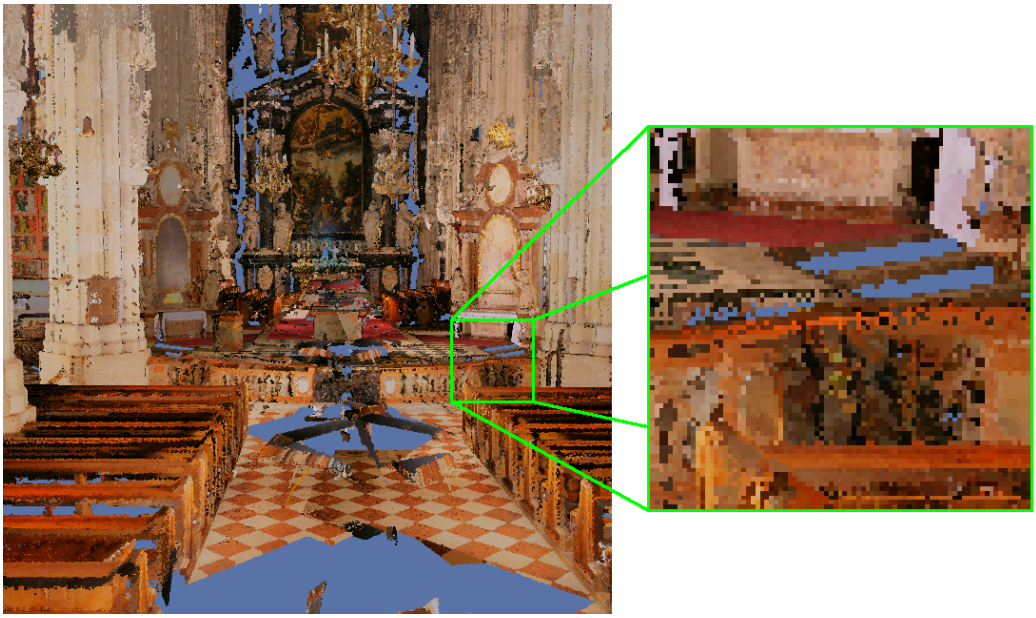

Fig. 6.12: The inside of the Stephansdom rendered with a minimum projected size for the MOSPT leaf nodes of 1 pixel. This is the highest available visual quality.

Fig. 6.13: Only relative few and large bounding boxes are visible, approximately 208. Almost all bounding boxes are in some shade of magenta, indicating that not all children nodes of these cells are rendered, because the projected sizes of the leaf nodes of the children are too small.

enabled or not. Occlusion culling uses the bounding boxes of the nodes to check if an MOSPT is occluded, so hidden bounding boxes will not be visible anyway.

In Table 6.5 the number of rendered cells and points are charted, when using different minimum projected sizes for the MOSPT leaf nodes. The maximum number of points that were allowed to be rendered was 12M, and 15M were allowed to be loaded to graphics card memory. In the first row this limit is almost reached. The loading was complete, because no better approximation to the final image was found with the available amount of points. When occlusion culling was enabled, the number of rendered points is about 1.5M smaller, but the number of points in the memory is the same. The decision if a point cloud is occluded is independent of the decision if the point cloud is loaded to memory. When the minimum projected size for the MOSPT leaf nodes is increased to 2 pixels, then the number of points loaded to memory is cut in half, for both situations, whether occlusion culling is enabled or not.

In Table 6.6 the values of some GPU performance counters are charted. The performance counters were read out using the NVIDIA Developer Control Panel, and an instrumented driver for the graphics card. Occlusion culling decreases the number of processed vertices per second (VPS) due to the latency when waiting for the occlusion query results. The number of points in relation to the number

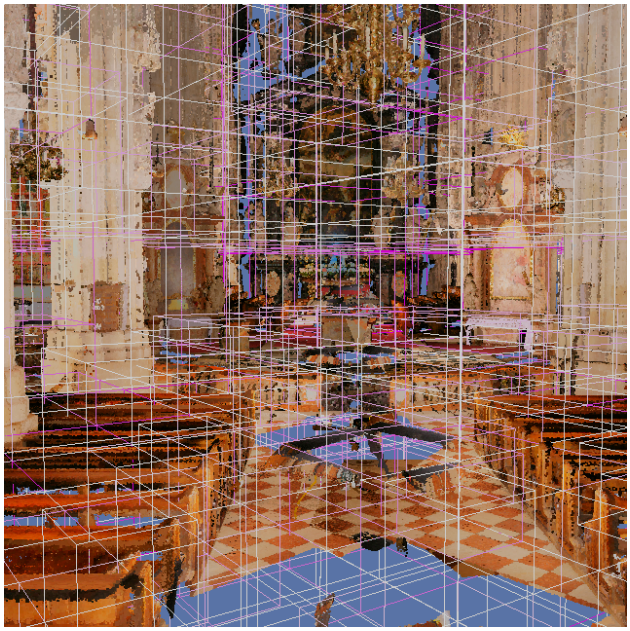

Fig. 6.14: Many bounding boxes are visible, approximately 499. About 50% of them are also white, which means all children nodes of these cells are rendered.

Parameters	Rendered Cells	Rendered Points
Min. proj. size 1 pixel	559	11,999,872
Min. proj. size 1 pixel, with OC	499	10,530,638
Min. proj. size 2 pixels	252	6,101,295
Min. proj. size 2 pixels, with OC	208	5,007,273

Tab. 6.5: The number of rendered cells and points. OC means occlusion culling. A decreased visual quality is traded for rendering fewer points.

Parameters	VPS	GPU idle	FPS
Min. proj. size 1 pixel	112M	0.04%	9
Min. proj. size 1 pixel, with OC	109M	0.20%	10
Min. proj. size 2 pixels	108M	0.06%	18
Min. proj. size 2 pixels, with OC	102M	0.50%	20

Tab. 6.6: The evaluation of performance counters on the GPU. Occlusion culling increases the latency when waiting for the results of the occlusion queries.

of pixels, used for the occlusion queries, is higher for a large model, so here the framerate for rendering the model with occlusion queries is above the framerate for rendering the model without occlusion queries. This is in contrast to the results for rendering a small model (see Table 6.2) where occlusion queries lowered the framerate. When using a threshold of 2 pixels for rendering, only half of the points are necessary as when using 1 pixel as threshold (see Table 6.5), and the CPU processing time during the rendering loop is too long to feed enough points. This is similar to Table 6.4, but when using a large model, the number of points that have to be rendered is much higher, so the GPU is better used to full capacity.

The following images from Figure 6.15 to Figure 6.20 were all taken with modified parameters. The resolution of the viewport was set to 1600x1200, the OpenGL field-of-view was set to 80 degrees, the number of points that were allowed to be rendered in one frame was set to 14M, and the minimum projected size for the MOSPT leaf nodes was set to 1. This results in a kind of “hi-res” images. From a blocky representation to the fully loaded model it took 35 seconds on the average.

In Figure 6.16 many empty regions are visible. The rhombus-like holes are at the places where the scanner was installed. At each of these places two scans were taken. The empty regions between the benches lie within the occluder shadow volume of the benches for the scanner position from which the point samples were taken. In the image the lighting can be seen in the regions where the point clouds of two scan positions overlap. The color of the floor is very dark, as the intensity of the light is falling off toward the border of the scanning area. The light intensity is also falling off toward the arches, so they appear too dark anyway.

Figure 6.17 is kind of a worst case scenario for the point rendering system, as many objects are visible near the viewpoint, and the model is stretching far into the distance. The maximum number of rendered points, 14M, is easily hit, and still some regions are not fully refined, as can be seen on the right side of the fence, where the LOD is too coarse for the distance to the viewpoint.

Figure 6.18 shows the lighting problems which appear when merging several different scan positions to one model. The altar on the right side in front of the benches is brighter than the altar on the left side. The lighting situation is not only

Fig. 6.15: A look through the Stephansdom from the main entrance.

changing within one scan, but especially between different scans.

Figure 6.19 the lighting problems are also visible. The altar in the left center of the image is lit very brightly, whereas the benches right in front of the altar appear very dark, because they were sampled from a different scan position. The statues in the upper regions of the columns are also sampled from different scan positions, and here the bright and dark samples overlap, giving the statues a spotted surface.

6.4 Summary

We tested the Nested Octree with a model consisting of 262M points. The model is built up from 77 different scan positions, where from each scan position only the contributing points were selected. The selection is done by oriented bounding boxes, which have to be defined by the user. The rendering system can achieve a high vertex throughput when using large models. Occlusion culling can be used to increase the framerate. When using small models that fit in the memory of the graphics card, also a high vertex throughput can be achieved, but occlusion culling should then be disabled. Interactivity is maintained at any time, even during loading new point clouds from the harddisk. The memory requirements of the outer octree are minimal compared to the size of the model, and so it can be kept in

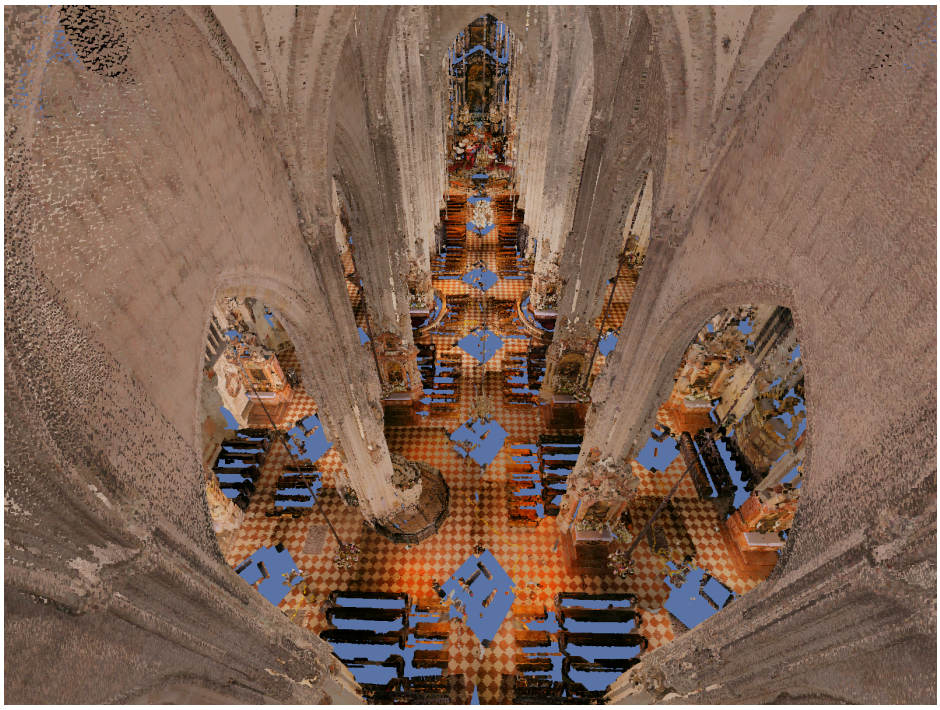

Fig. 6.16: Hovering inside the Stephansdom just below the arches.

Fig. 6.17: The inside of the Stephansdom as seen from the eye level of a visitor.

Fig. 6.18: The view from the center of the Albertinian choir to the high altar.

Fig. 6.19: Three altars in the northern part of the Stephansdom's nave.

main memory. The visual quality provided by the LOD algorithm is quite good, although undersampling occurs at the coarser levels of detail. If it is assumed that only minimal attributes are available for each point in the original point cloud, i.e., position and color, then the model can be rendered with good visual quality.

Fig. 6.20: The Leopold altar as seen from the scanner. Therefore the geometry in the image does not show large holes. On the right side of the floor the overlap of two scanning positions can be seen, where dark pixels from each of the two scan positions are in the direct neighborhood of bright pixels respectively from the other scan position.

Chapter 7

Conclusion and Summary

In this diploma thesis a new rendering system for point-based models was presented, which includes hardware-accelerated rendering, occlusion culling, and an out-of-core rendering algorithm. It is especially well suited to build up and render unprocessed point clouds, where only the position and optionally a color are available at each point.

7.1 Features

The rendering system takes point clouds as input. The only attribute that is necessary per point is its position in space. Using also color is however advisable. The point clouds can come directly from the output of a range scanner. Using oriented bounding boxes (OBBs), it is possible to compose a large point-based model from the point samples of several different scan positions.

The points from the original point clouds are then sorted into a novel data structure called Nested Octree (see Chapter 4). A Nested Octree is an octree where each node has an MOSPT associated with. An MOSPT (see Chapter 3) is a sequentialized hierarchy that can be processed completely on the GPU, and which uses only original points to provide a LOD mechanism. For rendering one LOD, all points from the upper levels are rendered down to the required level, and together the levels represent a certain LOD. All points that are processed and sent to the GPU are also rendered. The MOSPT alone allows for selecting one LOD at a time, but it does not allow for view-frustum culling.

The Nested Octree is used to provide view-frustum culling and out-of-core rendering. Each node in the Nested Octree contains the necessary information to render the associated MOSPT, like the number of hierarchy levels and the bounding box of the MOSPT. The MOSPTs from different nodes in the Nested Octree can overlap in space. The nodes to render are chosen by a benefit value, which depends on the relative position of the node to the viewpoint. If a node is chosen for rendering, than a request is sent to a second thread, which manages the loading of point clouds into memory. The point size that is used for rasterizing

a point on screen depends on the projected size of cells at the lowest level in the MOSPT. The same point size is also used by all parents of the node, if the node is at the lowest level in the hierarchy that is actually rendered. This ensures, that the point size is always as large as the finest available LOD for a branch in the Nested Octree hierarchy.

The rendering system is mainly focused on the fast rendering of point clouds, and not on rendering high-quality images of point-based models. This is on the one hand necessary, because we do not assume normals for the models, on the other hand it is intended as a fast reviewing algorithm for sampled point clouds, where the models can be reviewed at lower visual quality, but soon after the scanning.

7.2 Conclusion

Having implemented a point-rendering system from the ground up, one of the most important parts was to find out about the characteristics of the graphics card and driver. We were not sure how to achieve to highest vertex throughput, because we could not reach the number of vertices per second that was advertised by NVIDIA. We finally asked NVIDIA what the theoretical performance is. So testing the graphics card is certainly a good idea to find out about the limits of the hardware. The testing for the memory layout of the VBOs took some time. We finally came up with the MOSPT data structure which seems to be the fastest way to render small point clouds. The MOSPTs have to be inserted into an outer hierarchy to allow for view-frustum culling.

In Figure 7.1, three alternatives of how to place MOSPTs within a hierarchy, e.g., an octree, are shown. The placement in the left hierarchy would seem straight forward, but it has the problem that the granularity of the LOD selection is too coarse, and so too many points are stored on the graphics card which are not needed for rendering. The granularity of view-frustum culling is also too coarse, so too many points have to be processed. Another possibility to place the MOSPTs is shown in the middle. Here the lower levels of the hierarchy are converted into MOSPTs. This works well for point clouds that fit completely in the memory of the graphics card. For larger models this cannot be done. The problem is that the upper levels of the hierarchy are still within a conventional hierarchy, and how should they be traversed? They could be traversed point by point, but for really large models the upper levels will contain many points, so this will be slow. Finally the third alternative shows the idea we developed further, where MOSPTs, which are stored in VBOs, overlap each other. Here the LOD selection is fine enough, so that most of the points that are stored on the graphics card are also rendered, and the view-frustum culling is good enough, so that most processed

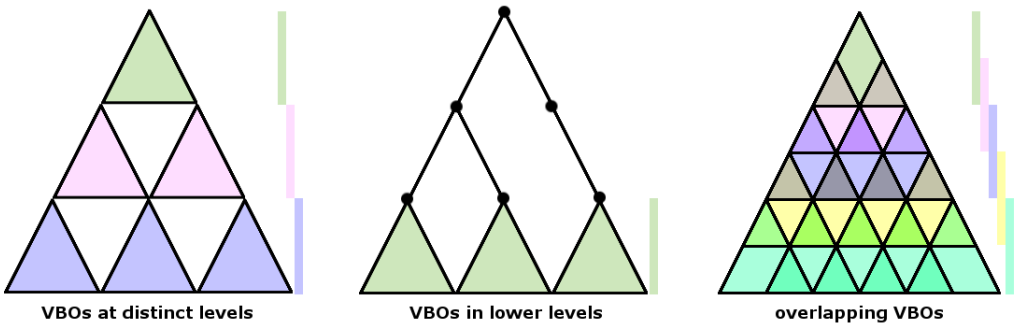

Fig. 7.1: Three alternatives how to convert an octree hierarchy to MOSPTs. Each MOSPT is stored in a VBO. The bar to the right of each hierarchy indicates the levels where the MOSPTs are placed.

points are actually inside the view frustum.

The next challenge was the out-of-core rendering and build up. The design and implementation of the system was not that hard, but the insight that the build up of the model and the rendering of the model, i.e., the loading of VBOs from disk to graphics card memory, takes much longer as when not dealing with out-of-core algorithms was striking. Theoretically the system can handle point clouds with 2^{31} points, but if we cannot find some optimizations for the build up process, testing this will be infeasible. The fewer data the out-of-core system has to process, the better. For even larger models than the Stephansdom, a compression algorithm would certainly be advantageous. The time necessary for compressing and decompressing the point attributes is probably less than the time it takes to store and fetch the uncompressed data from disk.

7.3 Possible Enhancements

The problems when rendering large point clouds are manifold. The amount of data that has to be processed during build up and rendering, the selection of points from the raw scanner data, the organization in a LOD hierarchy, the parameters during scanning, the selection of points for rendering, and so on.

For out-of-core rendering the number of disk accesses is the main performance criterion, since it takes an order of magnitude longer to fetch data from disk than to have it in main memory or even graphics memory ready for rendering. The time to load the point clouds from disk is not going to decrease soon, although the main memory and graphics memory are certain to increase on the average computer every year. So the number of points that can be stored in memory is going to increase, which will result in an improved quality for the rendered images. On the other hand, the models will continue to grow, consisting of 1 billion points

or more, so there is likely always the need for a fast out-of-core point rendering algorithm.

When composing models from several different scan positions, the lighting of the scans can vary widely, resulting in heterogeneously lit models, where the different scanning positions can be easily identified. One possibility to address this problem is the use of *high dynamic range imaging* (HDRI) during the sampling process. HDRI was first introduced by Ward in the Radiance rendering system [War94]. Debevec [DM97] developed it further, showing how HDR images can be created from several photographs of the same scene, where each photograph is taken with different exposure settings. The difference between HDR images and normal photographs is that HDR images store physical luminance values, whereas normal photographs only store a color value. If the lighting situation is known when taking HDR images, it might be possible to calculate a uniform lighting for the model, or to re-light each point to simulate different lighting situations.

The current rendering system is completely point based. This is the fastest method when converting raw data from range scanners to viewable models. To increase the rendering speed further, some more postprocessing of the raw data is necessary. Taking the model of the Stephansdom as example, there are some regions that could be very well approximated by polygons. Obvious choices are the floor as a whole, and the walls in part, as they are intersected by columns. The floor could even be textured by only one pattern. It is also scanned very densely, as it is close to the scanning positions. The speedup for using polygons instead of points should be quite noticeable. Even when using polygons, a LOD will have to be considered. To optimize the selection of the areas that should be replaced could be done interactively, although automatized systems already provide good results [WGK05]. The rendering system would then be implemented as *hybrid rendering system*, where models can consist of only points, only polygons, or both, similar to the rendering system of Chen and Nguyen [CN01].

The build up process is automated, after the user has defined an appropriate OBB for each scan position. Nevertheless, the build up process is time-consuming, so a possible enhancement could be an *incremental build up* of the Nested Octree. This means that a new scanning position can be included in an already existing Nested Octree, and the new points are filled into the hierarchy. The problem is that the leaf nodes of the Nested Octree can contain more points than necessary, so they should be re-opened and their points should be inserted again.

When scanning the environment, there will likely be areas that cannot be seen by the scanner's laser, because they are occluded by some other object. Another problem are areas where the sampling density varies due to the different number of scans that sampled the areas. A very densely sampled area can be adjacent to a poorly sampled area. A possibility is to *reconstruct* the model in these areas,

which could be done by a set of tools that interactively change the model. It will certainly be interesting to see whether editing can be done with really large models that do not fit completely in main memory.

Since the models are ever increasing in size, a possibility is to compress the point attributes, like position and color. There are two different methods of compressing, lossy and lossless *compression*. Higher compression rates are achieved with lossy compression, but this also requires the models to be resampled. Lossless compression on the other hand preserves the original points. Considering the increasing size of models, a compression of the attributes would be very interesting, especially if the decoding can be implemented directly on the GPU [KSW05].

7.4 Summary

The newly developed point-rendering system is well suited for building up models directly from the raw data of range scanners and rendering them fast, using the hardware-accelerated rendering pipeline. The only thing the user has to do is to define an OBB for each scan position, which is needed to find the desired points from the raw data of the scanner. Future enhancements could include an incremental build up of the hierarchy, compression of point attributes, reconstruction of the model, or a hybrid rendering system for using polygons.

Appendix A

Class Diagram

The point rendering system is implemented in the Yare Rendering Engine, which was developed by the real-time rendering group at the Institute of Computer Graphics and Algorithms at the Vienna University of Technology. The point rendering system includes several different algorithms, i.e., the ρ-grid, the SPT, the SPTs in an octree hierarchy, the MOSPT, the MOSPTs in an octree hierarchy, and the Nested Octree algorithm. For accessing the raw data of the point scans, the riscanlib from Riegl is used. The programming language for the implementation is C++.

The class layout for the Nested Octree algorithm is mainly aimed to support the outer and inner octrees. The main classes are:

- *GraphicsLruCache* manages the point clouds in graphics card memory. The LRU strategy is very simple, as all point clouds can be replaced, only the point clouds that were needed for rendering in the last frame cannot be replaced.

- *LruCache* manages the point clouds in main memory. The cache is simply a list where new elements are appended to the back. The first entry in the list is the oldest one. Elements can also be requested directly from the list. The second thread asks at first if a point cloud is stored in the LRU cache, and only if the point cloud is not available there, it loads the point cloud from the harddisk.

- *NestedTrigrid* builds up the runtime structure of the Nested Octree, and manages the rendering. The name Nested Trigrid is referencing some of the ideas we had during development.

- *NestedTrigridFile* handles the build up of the MOSPTs and the outer octree structure. For this it needs a stream of points, which is provided by the ScannerFileConverter class.

- *NestedTrigridSecondThread* is responsible for delivering point clouds from the harddisk or main memory cache to the graphics card memory. The communication between the threads is accomplished by a message interface. If the rendering thread needs a new point cloud, the necessary information is dispatched in a message. After loading the point cloud from harddisk, the second thread sends the information where the point cloud is stored in memory back to the rendering thread. But also other messages can be sent, like the number of points contained in the main memory cache.

- *ScannerFileConverter* can read a list of scanner files, and then provide them as a stream of points, which can then be processed by the NestedTrigridFile class.

- *TestApp*, which is an application that instances the scenegraph. The Nested Octree is attached to the scenegraph as a geometric object.

In Figure A.1 the UML class diagram of the Nested Octree point-rendering system is shown.

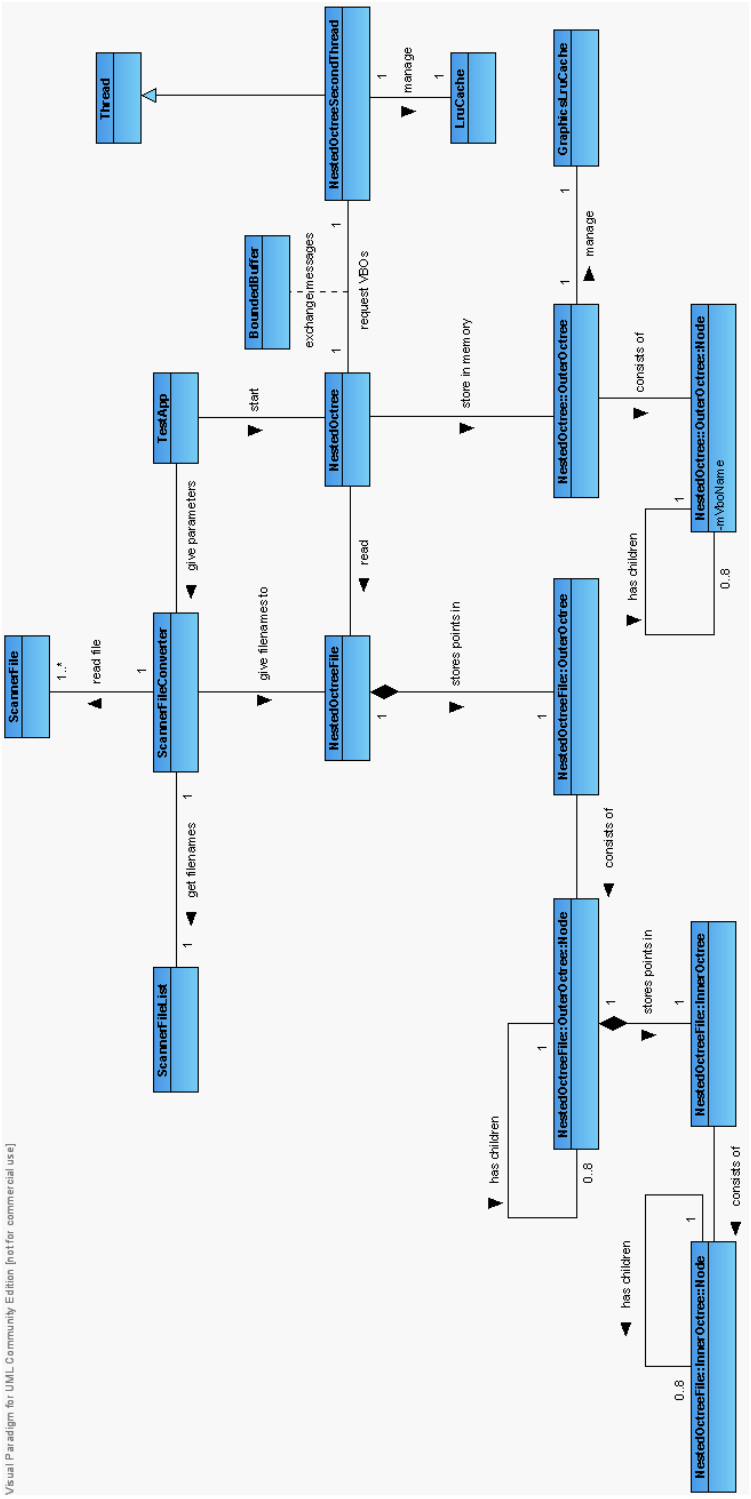

Fig. A.1: The UML class diagram of the implemented Nested Octree.

List of Figures

List of Tables

Bibliography

[BDS05] T. Boubekeur, F. Duguet, and C. Schlick. Rapid visualization of large point-based surfaces. In *Proceedings of VAST 2005*, pages 75–82, 2005.

[BHZK05] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. High-quality surface splatting on today's GPUs. In Marc Alexa, Szymon Rusinkiewicz, Mark Pauly, and Matthias Zwicker, editors, *Symposium on Point-Based Graphics*, pages 17–24, Stony Brook, NY, 2005. Eurographics Association.

[BR93] Jules Bloomenthal and Jon Rokne. HOMOGENEOUS COORDINATES. Technical Report 93/516/21, University of Calgary, March 1993.

[CN01] Baoquan Chen and Minh Xuan Nguyen. POP: a hybrid point and polygon rendering system for large data. In Thomas Ertl, Ken Joy, and Amitabh Varshney, editors, *Proceedings of the Conference on Visualization 2001 (VIS-01)*, pages 45–52, Piscataway, NJ, October 21–26 2001. IEEE Computer Society.

[Com06] CompuPhase. Colour metric. *http:// www.compuphase.com/ cmetric.htm*, 2006.

[dB00] W. de Boer. Fast terrain rendering using geometrical mipmapping, October 31 2000.

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

[DM97] P. E. Debevec and J. Malik. Recovering high dynamic range radiance maps from photographs. In *SIGGraph-97*, pages 369–378, 1997.

[DVS03] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. *ACM Trans. on Graphics*, 22(3):657–662, 2003.

[FS93] T. A. Funkhouser and C. H. Sequin. Adaptive display algorithm for interactive frame rates during visualisation of complex virtual environments. *Proceedings of SIGGRAPH'93*, pages 247–254, 1993.

[GD98] J. P. Grossman and William J. Dally. Point sample rendering. In George Drettakis and Nelson Max, editors, *Rendering Techniques '98*, Eurographics, pages 181–192. Springer-Verlag Wien New York, 1998.

[GM04] Enrico Gobbetti and Fabio Marton. Layered point clouds. In *Eurographics Symposium on Point-Based Graphics*, pages 113–120, 2004.

[Hec89] Paul S. Heckbert. Fundamentals of texture mapping and image warping. Master's thesis, CS Dept, UC Berkeley, May 1989.

[KSW05] Jens Krüger, Jens Schneider, and Rüdiger Westermann. Duodecim - a structure for point scan compression and rendering. In *Eurographics Symposium on Point-Based Graphics*, pages 99–107, 2005.

[LE97] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH 97 Conference Proceedings*, pages 199–208, 1997.

[LH04] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. on Graphics*, 23(3):769–776, 2004.

[LW85] 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.

[MCC+99] A. A. Mirin, R. H. Cohen, B. C. Curtis, W. P. Dannevik, A. M. Dimits, M. A. Duchauneau, D. E. Eliason, D. R. Schikore, S. E. Anderson, D. H. Porter, P. R. Woodward, L. J. Shieh, and S. W. White. Very high resolution simulation of compressible turbulence on the IBM-SP system. In *Proceedings of Supercomputing'99 (CD-ROM)*, Portland, OR, November 1999. ACM SIGARCH and IEEE.

[Mic03] Microsoft. How NTFS works. *Microsoft TechNet Webpage*, 2003.

[PSL05] Renato Pajarola, Miguel Sainz, and Roberto Lario. Xsplat: External memory multiresolution point visualization. In *Proceedings IASTED Invernational Conference on Visualization, Imaging and Image Processing*, pages 628–633, 2005.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proc. ACM SIGGRAPH 2000*, pages 335–342, 2000.

[RB93] Jarek Rossignac and Paul Borrel. Multi-resolution 3D approximations for rendering complex scenes. In *2nd Conf. on Geometric Modelling in Computer Graphics*, pages 455–465, 1993.

[RL00] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proc. ACM SIGGRAPH 2000*, pages 343–352, 2000.

[SS05] Oliver Schall and Marie Samozino. Surface from scattered points: A brief survey of recent developments. In Bianca Falcidieno and Nadia Magnenat-Thalmann, editors, *1st International Workshop on Semantic Virtual Environments*, pages 138–147, Villars, Switzerland, 2005. MIRALab.

[TKDS05] G. Li T. K. Dey and J. Sun. Normal estimation for point clouds : A comparison study for a voronoi based method. In *Eurographics Symposium on Point-Based Graphics*, pages 39–46, 2005.

[War94] Gregory J. Ward. The RADIANCE lighting simulation and rendering system. In *SIGGRAPH*, pages 459–472. ACM, 1994.

[WGK05] R. Wahl, M. Guthe, and R. Klein. Identifying planes in point-clouds for efficient hybrid rendering. In *13th Pacific Conference on Computer Graphics and Applications*, 2005.

[WS06] Michael Wimmer and Claus Scheiblauer. Instant points. In *Proceedings Symposium on Point-Based Graphics 2006*, pages

[ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 371–378, New York, NY, USA, 2001. ACM Press.