The Visual Computer manuscript No.
(will be inserted by the editor)

Jens Krüger · Jens Schneider · Rüdiger Westermann

Compression and Rendering of Iso-Surfaces and Point Sampled Geometry

Abstract In this paper we present a streaming compression scheme for gigantic point sets including per-point normals. This scheme extends on our previous Duodecim approach [KSW05] in two different ways. First, we show how to use this approach for the compression and rendering of high-resolution iso-surfaces in volumetric data sets. Second, we use deferred shading of point primitives to considerably improve rendering quality. Iso-surface reconstruction is performed in a hexagonal close packing (HCP) grid, into which the initial data set is resampled. Normals are resampled from the initial domain using volumetric gradients. By incremental encoding, only slightly more than 3 bits per surface point and 5 bits per surface normal are required at high fidelity. The compressed data stream can be decoded in the graphics processing unit (GPU). Decoded point positions are saved in graphics memory, and they are then used on the GPU again to render point primitives. In this way high quality gigantic data sets can directly be rendered from their compressed representation in local GPU memory at interactive frame rates (see Figure 1).

1 Introduction

Despite the advances in CPU and graphics hardware technology, for large iso-surfaces and point scans, point based rendering applications still cannot run at acceptable rates. As rendering capabilities continue to increase, so do the sizes of data being visualized as well as the resolutions of displays being used. Today, iso-surfaces of volumetric data, numerical simulations, and laser range scans [Lev00, LPC*00] produce gigantic data sets, that can result in billions of vertices, making even CPU processing difficult due to memory constraints. Figure 2 shows such gigantic data sets. A triangulated iso-surface of the largest of these data set consists of about one billion vertices and normals and requires 24 GBytes for geometry alone. Because of the extraordinary richness of detail in these data sets, the need for techniques able to reveal even the finest structures is becoming increasingly important. In addition to such data sets, high resolution display systems of over 10 Mpixels [IBM] are nowadays available, letting the required bandwidth to transmit primitives to the GPU grow substantially. As these requirements will continuously increase in the future, there is a dire need for point rendering techniques that comprehensively address these issues.

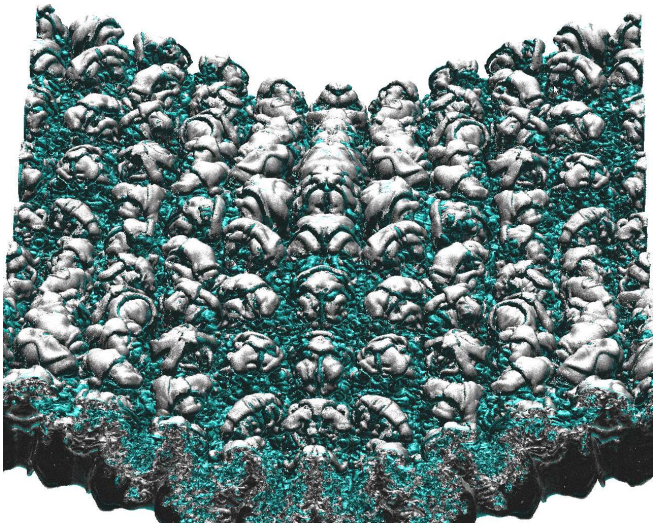

Fig. 1 A time step of the mixing fluid simulation. In this image two iso-surfaces (cyan and white) are rendered simultaneously. A Marching cubes iso-surface extraction of these surfaces results in about 1 billion points consuming 24 GB for the point and normal information alone not considering connectivity. However the two compressed iso-surfaces rendered here require only about 500 MB. Data set courtesy Lawrence Livermore National Laboratory.

The original publication is available at www.springerlink.com

Computer Graphics and Visualization Group, Technische Universität München
Tel.: +49-89-289-19461
Fax: +49-89-289-19462
E-mail: {jens.krueger jens.schneider westermann}@in.tum.de

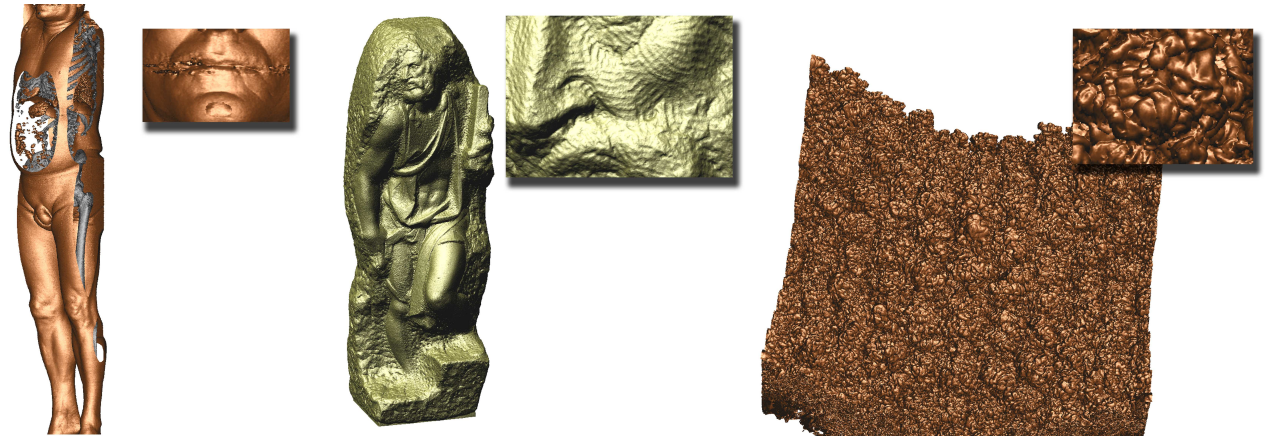

Fig. 2 Two large iso-surfaces and one point scan are shown. All three data sets, including per-point normals, have been compressed and now fit into 256 MB video memory on recent graphics cards. Images are generated by rendering the scans out of the compressed data stream on the GPU. Up to 50 million points per second can be decoded *and* rendered on consumer class graphics hardware. Data sets courtesy of Siemens Corporate Research, The Digital Michelangelo project, and Lawrence Livermore National Laboratory.

In computer graphics, point based rendering has recently gained increasing popularity due to the simple and memory friendly nature of point rendering primitives. Such primitives do not require consistent topological information and they considerably reduce overdraw if high resolution models are rendered. A thorough discussion of these issues as well as a summary of recent point rendering techniques including various applications can be found in [GPA*, KB04].

First considered by Levoy and Whitted [LW85] and then revived by Grossmann and Dally [GD98], rendering systems based on point primitives have been proposed for both the hierarchical display of large point scan models [RL00] and high quality rendering of point sampled geometry [PZvBG00, ZPvBG01]. Due to the frequent use of such systems in practical applications, over the last few years there has been an ongoing improvement in this field with respect to rendering speed and quality, i.e., by exploiting graphics hardware [RPZ02] and efficient GPU data structures [DVS03], by using high-quality point splats [BK03, ZRB*04], and through the use of point hierarchies [GM04] in combination with polygonal mesh representations [CN01, CAZ01, DH02, GM05] to allow for efficient LOD rendering.

Polygonal iso-surface extraction and rendering on the other hand has position itself as a powerful tool for interactive exploration of large volumetric datasets for almost three decades now. Approaches to handle large data sets that have been developed since the early Marching Cubes paper [LC87] can be categorized roughly to fit into one of the three categories:

Acceleration with hierarchical data structures: To speed up the rendering of iso-surfaces by considering only relevant parts of the data set, hierarchical data structures such as the Octree [Lev90] or span space trees [LSJ96] have been used.

View Dependent Iso-Surface Reconstruction: These algorithms generate isosurfaces on-the-fly using a view dependent error measure. This dramatically reduces the amount of data to be considered for isosurface extraction and the amount of geometry to be rendered [LH98].

3D Texture based iso-surface reconstruction: In texture based iso-surface extraction instead of generation the iso-surface's geometry explicitly a shader program or a special GPU configuration was used to render only parts of the volume that do belong to the surface [WE98] normals to compute the lighting where generated on the fly by a accessing an additional gradient volume.

Although all of these methods have undergone numerous improvements in the past 10 to 20 years interactive rendering of gigantic data sets of our target size still remains a huge challenge. This is simply due to the sheer amount of data that would have to be processed out of core not only during preprocessing but during rendering as well for most of these algorithms.

In the same way today's point rendering systems are facing the problem of continually increasing point sets. To keep up with this progress, several issues have to be considered: For large point scans the CPU might not be equipped with sufficient system memory. If the CPU works on a compressed data set, it might not be powerful enough to decode point positions and attributes at sufficient rates. If, on the other hand, a streaming representation is available that enables out-of-core rendering, disk access will most likely limit the overall performance. Even if the CPU could provide point rendering primitives at sufficient rate, the bandwidth of the communication channel connecting the CPU with the GPU might be too low as to allow for the transfer of point positions and attributes a fixed number of times per second. Finally, the GPU itself might not be able to render the points within the requested time interval.

Up to now, only a few approaches have focussed on these issues explicitly. A popular technique is to quantize point positions with respect to a Cartesian grid hierarchy, either by absolute position encoding or relative to a parent node in this hierarchy [RL00, SK01, BWK02]. Although these approaches can significantly reduce the required number of bits to encode point positions, a similar compression ratio has not yet been shown for normals. Ochotta and Saupe [OS04] locally parameterized point sets as a hight field and resampled the point sampled surface on a regular grid. This method achieves high compression ratio by using wavelet transforms to encode the resulting height fields, but it introduces additional smoothing artifacts and produces a non-uniform sampling of the surface. A progressive compression scheme for point sets including per-point attributes based on multiresolution predictive encoding was presented by Waschbüsch et al. [WWL*04]. This scheme yields effective compression rates, but it suffers from both the high complexity of the matching process to find similar points and the rather costly decoding process, which recursively traverses binary trees to calculate initial point positions.

In this paper we present a streaming compression and rendering system for gigantic point sets that comprehensively accounts for the aforementioned requirements. This work extends our previous Duodecim approach [KSW05]. In contrast to previous compression schemes for point sets, point coordinates can be decoded on the GPU. Our scheme has the following particular properties:

- **Memory efficiency**: We present an effective compression scheme for large iso-surfaces and point scans based on an optimal sampling of these data sets.
- **Decoding efficiency**: The compressed stream provides random access to encoded points and attributes, and it can be decoded using a few simple arithmetic and logical operations.
- **Bandwidth efficiency**: Due to its simplicity, decoding can be performed on the GPU. To render the point set only the compressed data stream has to be transmitted.
- **Rendering Efficiency**: On the GPU, decoded point positions and normals are used in turn to render the point scan, which results in a significant performance gain compared to previous approaches.
- **Variable Rendering Quality**: While the straight forward point rendering mode gives the user maximum rendering performance he can also choose a high quality mode in which our system uses deferred shading similar to Botsch et al. [BHZK05] to generate higher quality images.

2 The DuoDecim Engine

To enable the aforementioned compression properties, we introduce closest sphere packing (CSP) grids as a new and effective spatial data structure for point clustering. From closest sphere packing theory [CSB87] we know, that an optimal sampling in the spatial domain corresponds to the tightest arrangement of spheres in frequency domain. This can be derived from the observation that the spectrum of the sampled signal contains the replicas of the primary spectrum, centered at the points of the dual (or reciprocal) of the sampling grid. Optimal sampling of the signal is achieved if there are no overlappings between the replicas.

The closest sphere packing (CSP) grids we employ are composed of Trapezo Rhombic Dodecahdra (TRD) - the dual of the Johnson Solid 27 [Joh66]. A TRD is a space filling twelve (twelve=duodecim (latin)) sided polyhedron, which constitutes a base element for a closest sphere packing of 3D space. CSP grids consisting of TRDs in particular (HCP), have no second order neighbors, i.e. no cells share only an edge. If such a grid is sliced orthogonal to the y-axis, the resulting 2D grid is composed of regular hexagonal cells. In this 2D grid, all neighbors share an edge and the distance between adjacent cell centers is constant. These properties are illustrated in figure 7. Note that besides the CSP grid we use, there exists the Face Centered Cubic (FCC) grid composed of the Rhombic Dodecahedron [Mat04, NM02]. Although both grids are closed sphere packing grids, HCP grids have some beneficial properties compared to FCC grids. Most importantly, the dual grid of FCC, the Body Centered Cubic grid (BCC), which is used for sampling, is not composed of a single cell type. Therefore the volume of the different sampling cells is distributed non-uniformly. This increases the maximum sampling error compared to HCP grids, although the same number of cells is required to closely pack 3D space. To our best knowledge, the HCP grid has never been employed in computer graphics applications so far.

We take advantage of HCP grids to establish an adjacency relation between points. Therefore, a binary spatial data structure consisting of TRDs is generated, where a cell is full if it contains a point, and it is empty otherwise. The adjacency relation is exploited to incrementally encode runs of connected full cells in slices of the grid. This stage is very similar to the process described in [MH01] for the encoding of iso-surfaces in volumetric data sets. In contrast, however, we turn the problem of determining optimal runs into a graph theoretical one, which is posed as a search problem to find the minimum number of paths of a specified length that cover an arbitrary graph.

In addition to incremental encoding of point coordinates, the same compression scheme can be applied to per-point attributes like normals or colors. Because these attributes show only a slight variation along the selected point runs, high fidelity at low bit rate can be achieved by differential encoding. To support view frustum and back face culling, run-specific attributes like cone of normals and bounding boxes are computed.

The compressed point set can be decoded on the CPU, and point primitives can be sent to the GPU for rendering. Alternatively, the compressed stream can be decoded on the GPU. In this case, runs of equal length are stored in 2D texture maps and can then be decoded incrementally. Decoded point positions are first saved in graphics memory, and they are then used on the GPU again to render point primitives. This is realized using recent functionality like vertex texture fetches in the Shader Model 3.0 [Mic04] or Render to Vertexbuffer functionality [ATI04]. For the rendering of large point sets that do not fit into graphics memory, the presented point rendering system can either avoid bus transfer at all, or it can reduce bandwidth requirements substantially if the meshes are so large that even in compressed format they do not fit in graphics memory.

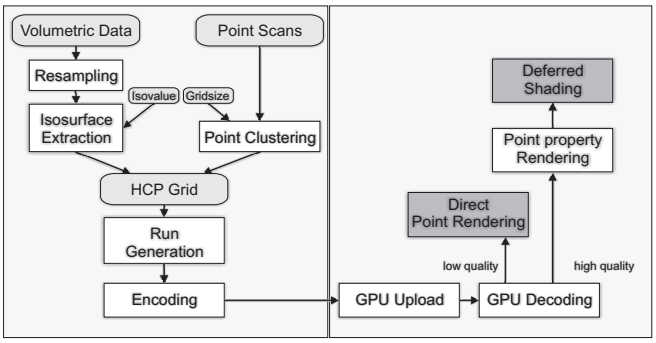

Fig. 3 The Duodecim Engine consists of two modules First the offline encoding part. This module takes either a volumetric or point data sets and compresses it into the DuoDecim format. The second online module produces either fast low quality or - with about half the frame rate - high quality images.

Since the input data is rarely already quantized into our HCP representation in general we need to convert it. Therefore we first apply resampling to volumetric data and grid clustering to points clouds (see Figure 3).

2.1 Iso-Surface Preprocessing

To improve image quality, processing speed, and compression ratio we do not extract the iso-surface in the original data volume but convert the volume into our our HCP representation. After the conversion we perform the iso-surface extraction in HCP space and compress this data. This guarantees optimal run connectivity and requires no further data transformation if the user changes the isovalue.

Compared to other grids, the HCP grid leads to an optimal sampling density, and in particular compared to Cartesian grids it yields a significantly smaller sampling error if a region in 3D space is partitioned using the same number of cells. In Cartesian grids with a grid spacing of h the maximum distance – and thus the upper bound for the sampling error – between a cell center and any other point inside a cell is $\sqrt{3}/2h$. For a HCP grid, on the other hand, it is only $\sqrt{3}/(2 \cdot \sqrt{2})h$.

Since optimal resampling to the HCP grid requires a spherical, bandwidth limited interpolation kernel [NM02], we use a spherical Lanczos kernel of radius ρ:

$$L_\rho(r) = \begin{cases} \mathrm{sinc}(r \cdot \pi/\rho) & \text{if } \mathrm{r} \leq \rho \\ 0 & \text{otherwise} \end{cases}$$

Note that the kernel still has to be properly normalized. In the case of rectilinear grids, we keep up to $2\rho+1$ slices in memory and perform the convolution with $L_\rho(r)$ in the spatial domain. For unstructured grids this involves a cell search, which is implemented efficiently using standard binary space partitioning approaches. Changing the radius ρ offers the user a speed/quality tradeoff.

The conversion form the input grid domain results in a HCP grid with a single data value for every grid vertex. To compress a user defined iso-surface from this representation the system selects the HCP cells with a sign change i.e. for all cells the values at the vertices are compared to the iso-value and any grid cell that has both vertex values above and below this user defined iso-value is selected.

Since the optimal HCP grid size is known a priori for iso-surface preprocessing, we can skip the costly grid size optimization and proceed directly to run generation 3.

2.2 Point Scan Preprocessing

To exploit geometric coherence in unstructured point sets, an adjacency relation between points is established first. We employ a regular spatial data structure composed of TRDs, into which the original point set is sampled. Every TRD that contains at least one point is marked with 1, while any other cell is marked with 0.

In the sampling process, the cell spacing of the HCP grid has to be determined such that as many cells as possible contain only one original point. On the other hand, by using ever smaller cells, connectivity between filled cells will be lost. This reduces efficiency of the incremental encoding step. Due to this reason, sampling point scans is implemented as an optimization problem that takes in account both constraints.

2.2.1 Sampling

To sample a point set, we start with an initial resolution of the HCP grid. In the current implementation this resolution is equal to the sample spacing of the scanning device. If this spacing is not known, an arbitrary initial guess can be specified. During the sampling process, we count for every cell how many points are sampled into

this cell. Let us call this value the *hit rate*. Now the initial resolution of the HCP grid is iteratively refined until the hit rate drops below a given threshold.

For the sake of simplicity we explain the algorithm for the 2D hexagonal slices, the extension to 3D HCP grids is canonical. A constant time sampling strategy can be derived from the following observation. Considering the odd and even rows separately generates two cartesian grids. The vertex in question is sampled into these two grids. This generates two cells that correspond to two hexagonal cell candidates. To choose from these cells the distance from the vertex to the two cell centers is computed and the cell with the smallest is the correct hexagon (see Figure 4). In 3D four cell candidates are generated and four distances are compared to find the smallest value. Hence, sampling into the HCP reduces to a fixed number of modulo operations and distance calculations.

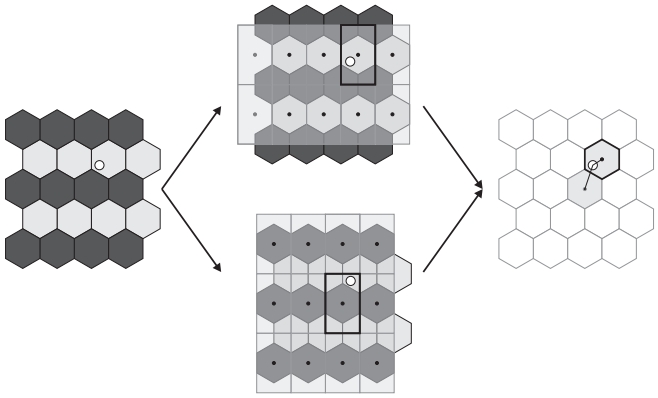

Fig. 4 This figure illustrates the sampling of points into hexagonal grids using two staggered Cartesian grids.

As the point sets and thus the grids we are concerned with are very large and can usually not be stored in main memory, the entire sampling process is performed out-of-core. Point subsets are sequentially sampled into the grid. For the laser scans, filled grid cells are then sorted on disk with respect to increasing cell index along the x-, z-, and y-axis. For volumetric data, this step is not necessary, since the volume can be resampled in an appropriate order. The sorted list can then be traversed sequentially to determine duplicate samples in one cell, and to compute the average hit rate.

This sorting step allows us to perform all following operations on a highly compact data window. This is in contrast to streaming approaches for triangle meshes, where topology in general prevents such a sorting. Thus, we can amortize the costly out-of-core sorting over all consequent operations in the pipeline.

At the end, the point set is implicitly given by a set of TRDs – or more precisely by the coordinates of their centers – that contain at least one point of this set. Thus if more than one input point falls into a cell only one representative (i.e. the cell center) for all of these points is stored.

2.3 Run Generation

The HCP grid provides a structure to generate contiguous runs of filled cells. This step is the transition from pure clustering to coherence based compression. The goal is to determine runs that are as long as possible, and to incrementally encode the cells in these runs. Starting with the grid index of the first cell in such a run, following cells can be encoded by storing the face they share with the predecessor. In this work, we restrict ourselves to the generation of runs within slices orthogonal to the grid y-axis. Because in such a slice every element only has 6 faces and a path never leaves through a face it just entered, adjacency information can be encoded in $\log_2 5 \approx 2.3$ bits.

Run generation proceeds layer by layer, reading all cells in the current layer from disk. By connecting these cells, an undirected graph is constructed. Run generation now operates on this graph, which makes the algorithm independent of the underlying grid structure. Ideally, the algorithm finds the minimum number of edge runs of given length *SL* that cover the entire graph and do not contain any edge twice. Since the solution to this problem is NP-hard, we present a linear-time 2-approximation, i.e., one that contains every cell at most twice. Note that this 2-approximation property is a very conservative upper bound. In all meshes we processed, the runs never contained more than 5 percent of all points more than once.

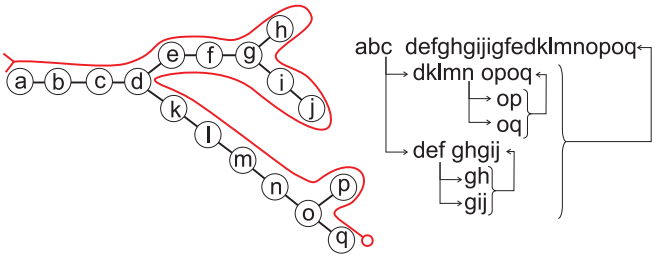

Fig. 5 The Round-Trip Generation

At first, for every connected subgraph a single run is generated, so-called long-runs, which are then cut into pieces of length *SL*. Starting with an arbitrary node in the graph, as long as this node has exactly one neighbor that has not yet been visited, the current node is appended to the long-run and the neighbor becomes the current node. If there are no such neighbors, the run is terminated. Otherwise, for every neighbor that is encountered a new long-run including the current node as start element is generated. For every new run but the longest one we now produce a round-trip, i.e. a run that

returns to its starting point. If all new runs, with the longest of these runs considered last, are appended to the current run, a contiguous path is generated that traverses every branch but the longest forth and back. What remains to be done is to eliminate redundant pieces, i.e. round-trips that begin a run and parts that appear in another short-run. Figure 5 illustrates this algorithm. For the algorithm we never need to consider more than three neighbors. Figure 6 shows that if more than three neighbors are filled, these additional neighbors are traversed by the child calls already and do not need to be traversed by the parent anymore.

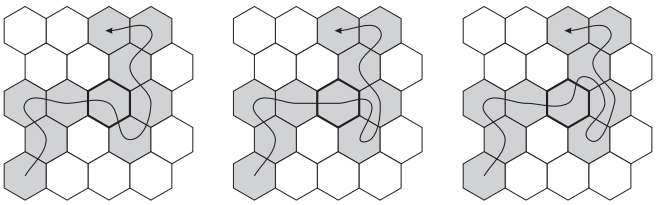

Fig. 6 The images show a cell with four neighbors. Note that independent of the choice of the first child to be traversed, all neighbors are handled before the recursion returns to the parent node. For more than four neighbors this procedure is alike.

Due to the construction of runs, in every long-run an index can appear at most three times. However, for an index that appears more than twice there always exist at least one index that appears only once. This property can be proved by the means of complete induction over all T junctions. If a child run at a T junction was chosen to be serialized, it must have been the shorter run of the cell. Therefore the cells of the longer child are not duplicated. This always duplicates less then half of the cells while only the T junction itself could be tripled. Consequently, the algorithm computes a 2-approximation of the optimal solution.

This procedure generates for every connected subgraph of one slice a single run. These long-runs are cut into pieces of given length SL; so-called short-runs. While this cutting takes place several optimizations are performed to further reduce the number of redundant cells. One of these optimization is to skip the way back of a previously serialized child run.

By restricting the maximum run length, the number of generated runs is increased at the same time decreasing the variation of their lengths. This kind of construction accommodates perfectly to GPU rendering in that SIMD streaming computations on such architectures can be exploited. If multiple processing units, i.e. fragment units, decode a number of runs in parallel, it is desirable for every run to contain exactly the same number of encoded points.

To generate a LOD hierarchy of the point set, sampling and run generation is repeated with decreasing resolution of the HCP grid. Starting with the optimal resolution, at every hierarchy level the resolution is reduced by a factor of two. Because the resolution at every coarser level is now fixed, grid size optimization of runs does not have to be carried out.

2.4 Optimization

In case our input data is a point scan we do not necessarily know the optimal grid size in advance. Therefore we employ an optimization algorithm on the grid resolution for sampling and run generation, we consider the average length of short-runs in addition to the average hit rate. The first value measures how many points are lost due to the sampling process. The second value is a measure of the compression efficiency. A perfect sampling would result in an average hit rate of 1 and an average length of short-runs equal to SL. Making the grid cells smaller results in a lower hit rate but reduces the average run length. To find the optimal cell size we first start with an initial size, which is 1.5 times the average distance of points in the point set, computed on a small set of representatives. If the average hit rate is above a limit, usually 1.6, the cell size is reduced according to ratio between the maximum hit rate and the current hit rate.

The sampling process is repeated with the new grid resolution, until the hit rate is below the maximum hit rate. Then, the run generation process is started. If the average length of short-runs is below a given threshold, usually 70% of the maximum length, we first try to close disconnected runs by inserting new cells. If this does not bring the average run length above 70%, the grid cell size is enlarged, sampling is repeated, and new runs are generated. The process terminates and outputs all current runs if the average run length is reached. Note that the 70% value is an arbitrary starting value for the algorithm based on our experience with different scans, even with other starting values the algorithm will still find an optimal value but it will possibly take longer to converge.

To close disconnected runs, we search for filled cells in the 2-ring neighborhood (see figure 7) of those cells that are contained in runs shorter than 50% of the maximum length. We do not try to connect longer runs because this could result in runs that are so long that they are cut into short runs later. If such a cell is found and both cells belong to different runs, the cell in between is set to connect the two runs. As the examples below demonstrate, this process adds far less than 10% of the initial cells on all models we tested.

2.5 Encoding

The proposed scheme generates a large set of runs less or equal to a given length. To every run, three 16 Bit values are associated: the first two values are used to encode the start cell of each run within the current slice.

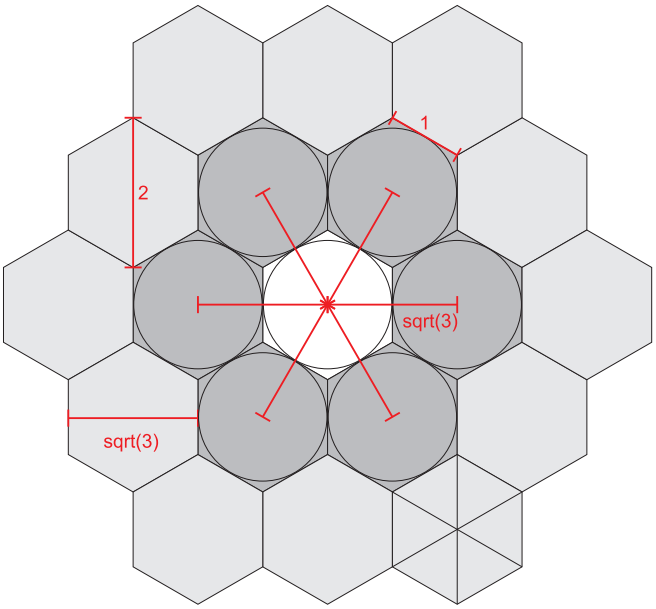

Fig. 7 Geometric properties of a hexagonal 2D slice of the HCP with the one and two-ring neighborhood

Positions in slices of a resolution of up to $2^{16} \times 2^{16}$ can thus be encoded. The third 16 Bit value is used as index into a codebook that contains quantized normals. The computation of this index as well as the codebook is described below.

2.5.1 Point Encoding

In a run, every point but the first one is encoded relative to its predecessor using 2.3 bits to encode a step to one of the 5 still unused adjacent neighbors. In our current implementation however, we use 3 bits per vertex to keep the decoding process as simple as possible, and thus to enable the GPU to efficiently decode point runs. If decoding is to be performed on the CPU, memory requirements can be reduced about 25% by using all bits in the data stream. On disk, we further reduce storage requirements by using an entropy coder. This exploits the fact that runs in flat regions produce uniform bit patterns. Using ZIP compression reduces the file size by about another 33% on average.

2.5.2 Normal Encoding

Normals are either given for the original point set, or they are computed prior to the compression stage, e.g., by computing normals on a given triangulation or by moving least squares [ABCO*01] or, in the case of volumetric data sets, normals are generated during the resampling process.

As adjacent normals in a run only show a slight variation, they can be encoded incrementally. This assumption may lead to a higher local error across sharp features where the normals are not continuous, however this error vanishes rapidly along the run and after the next run restart. Every normal but the first one is expressed in spherical coordinates relative to its predecessor. Let θ and ϕ be the azimuth and the longitude coordinates, respectively, of the current normal. To avoid suboptimal compression at the poles we compute both the negative and the positive angles and use the one that leads to a smaller delta. If the difference to the following normal in spherical coordinates is $\Delta\theta, \Delta\phi$, then the new normal in Euclidean space coordinates is given by:

$$x = cos(\theta + \Delta\theta) \cdot sin(\phi + \Delta\phi)$$
$$y = sin(\theta + \Delta\theta) \cdot sin(\phi + \Delta\phi)$$
$$z = cos(\phi + \Delta\phi)$$

As this computation requires trigonometric functions to be evaluated, we employ trigonometric relations

$$sin(\alpha + \beta) = sin(\alpha)cos(\beta) + cos(\alpha)sin(\beta)$$
$$cos(\alpha + \beta) = cos(\alpha)cos(\beta) - sin(\alpha)sin(\beta)$$

to express Euclidean space coordinates in terms of precomputed sine and cosine values. More precisely, given $sin(\Delta\theta)$, $cos(\Delta\theta)$, $sin(\Delta\phi)$ and $cos(\Delta\phi)$, as well as the respective values for the previous normal, Euclidean coordinates for the current normal can be decoded using simple arithmetic.

During run generation, we collect all normal increments in spherical coordinates that occur in the entire data set. These increments are then clustered using vector quantization [AG92]. The two angular increments in the codebook are stored as four sine and cosine values for each entry. To avoid accumulation of quantization errors, each normal of a run is encoded relatively to its reconstructed predecessor. In the current work, 16 bits are used to quantize the start normal of every run, and 5 bits are used to quantize normal increments.

2.6 Rendering

To render the compressed point set, the encoded data is traversed slice by slice. For each run, the start position is decoded from the associated grid coordinate, and the start normal is fetched from the quantization codebook. All other point coordinates can then be decoded incrementally from the relative offsets that are stored with respect to the underlying grid structure. Only for the incremental decoding of normals an additional lookup into the delta normal codebook is required. This process consecutively generates pairs of coordinates and normals, which are written to a vertex and a normal array. Via graphics APIs like OpenGL or DirectX, these arrays can then be issued for rendering.

If decoding is carried out on the CPU, the vertex and the normal array have to be sent to the GPU, making bus bandwidth a major bottleneck. To overcome this

limitation, encoded runs are sent to the GPU in compressed form. Due to the simplicity of the decoding process, runs can be directly decoded on the GPU using parallel streaming computations. Reconstructed point positions are rendered directly without any read back to application memory. In this scenario, the CPU is only used to control which runs are sent to the GPU, i.e. to accommodate view frustum and backface culling (see below).

The GPU decoder exploits functionality on recent graphics cards. On such cards, it is now possible to access texture maps in the vertex units [Mic04] and to render into special texture surfaces that can be interpreted as textures and vertex buffers alternatively [ATI04]. Our rendering system exploits both functionalities on both graphic APIs, DirectX and OpenGL. This leads to four different render backends from which the user can choose

To prepare compressed point runs for GPU processing, they are stored in 2D texture maps. For each run, its start position and normal is stored in a 16 bit RGB texture map. Consecutive points in a run are encoded in 8 bit luminance textures, using the first 3 bits to store adjacency information and the remaining 5 bits to store quantized delta normals. These normals are decoded from a quantization codebook. The principal layout of all data structures on the GPU is illustrated in figure 8.

In consecutive rendering passes, the GPU first decodes per-point normals and renders these normals into an intermediate memory object, i.e. a normal map. Then, point positions are decoded, the normal map is read to enable lighting computations, and positions as well as computed colors are rendered into a second memory objects. This object is then bound as a vertex array and can thus be rendered without any read-back to the CPU. In the following pass, both intermediate memory objects are read to obtain previous point positions and normals required for incremental decoding. Then, GPU decoding proceeds as described. Thus the rendering all runs consisting of n takes n rendering passes.

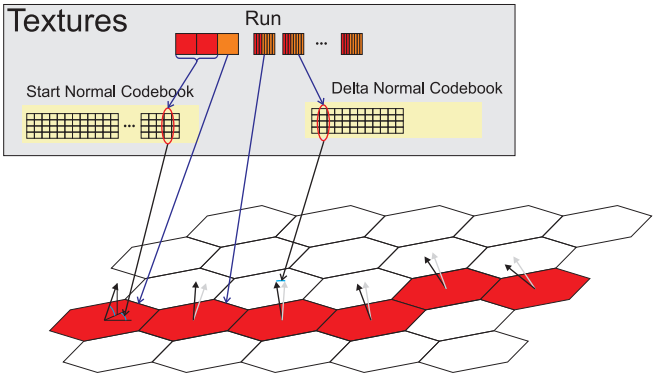

Fig. 8 This image illustrates the encoding of runs into texture maps on the GPU.

2.6.1 Deferred Shading

In maximum performance rendering mode the DuoDecim engine uses the decoded points and normals to computes phong illuminations in the vertex shader. Although this approach is very fast it results in a per splat constant color shading. Especially in closeups when the splat size becomes larger this causes disturbing artifacts (see Figure. 9 left).

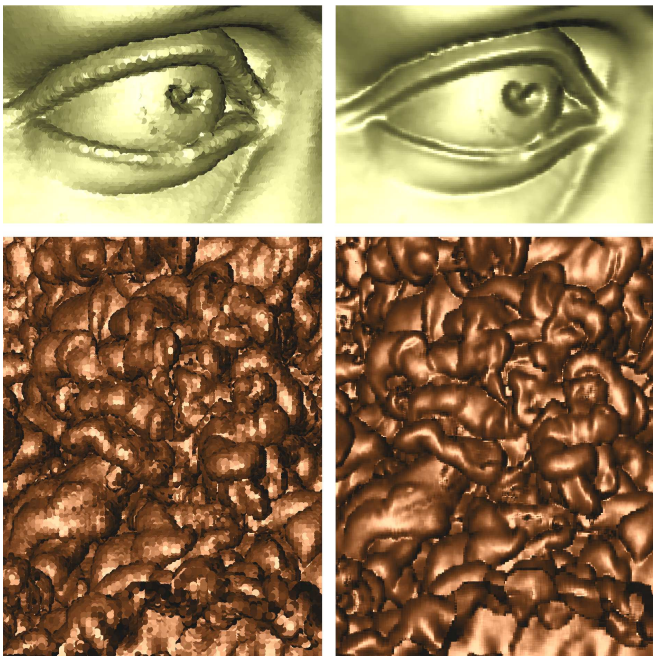

Fig. 9 Comparison of per splat (left) and per pixel phong illumination (right). The upper images show a close up of David's mouth while the lower to image show an extreme zoom into the fluid mix data set

To eliminate this problem a per-pixel lighting is required. As elaborated in Botsch et al. [BHZK05] deferred shading promises the best performance on current GPUs. To compute this per pixel lighting we accumulate the splat normals of the visible surface in the frame buffer. In the fragment stage the engine weights the normal of every fragment with its distance to the splat center. Since this accumulation requires alpha blending we can not use the z-test and have to do a depth pass first to determine the visible splats. Furthermore current GPUs do not support full 32bit floating point blending therefore we write the normals into a 16bit floating point target. After we have completed the generation of the normal image we compute the illumination per image pixel (see Figure 10).

Since we need to render the points twice and do not assume to have enough GPU space for decoding the entire data set, we have to decode every point twice and execute the following multipass algorithm:

Fig. 10 The three passes needed for our deferred shading approach. The first pass generates a depth image. The second generates the point properties - the image shows only the normals. During this second pass the depth values from the first pass are used for a depth equal test. In the third image-based pass the results from the second pass are used to illuminate the final image.

Depth Pass: For every visible section of the data set decode the point position and write the depth values into a single component floating point render target. During this pass the z-test is enabled, thus leaving only the front most depth values in the render target.

Property Pass: For every visible section of the data set decode the point position and point normals and write the normals into the three component float 16 target. In the pixel shader the the depth value of every fragment is compared to the depth value in the z-texture from pass 1. If this test indicates that the fragment lies more than epsilon behind the textures depth value the fragment is discarded. For all other fragments the transparency is set in accordance to their distance from the splat center. If other per splat values such as color or texture coordinates are present they can be written into multiple rendering targets in this pass. And do not require additional passes. During this pass the depth test is disabled and blending is set to add the values in the framebuffer to the incoming values.

Illumination Pass: In the final pass a single quad covering the entire viewport is rendered. This quad generates a fragment for every pixel on screen. In the pixel shader for every fragment the accumulated normals are read from the texture generated in the second pass. This normal is renormalized and used for illumination. If other properties where generated in the second pass as well they are now read too and contribute to the final color. In this pixel shader any illumination scheme can be used such as simple phong illumination, toon-shading, or edge enhancement shaders etc. During this pass no fragment tests or blending parameters are enabled.

This deferred rendering scheme reduces the rendering performance by a factor of two since every point needs to be decoded and data needs to written to floating point targets. On the other hand as can be seen in Figure 9 the smoother shading dramatically improves the image quality.

2.6.2 Culling and LOD

To render the compressed representation, the CPU determines the runs to be rendered, and it sends the textures containing these runs to the GPU. To keep bus transfer and GPU processing minimal, two different acceleration techniques have been integrated into our approach: First, runs are clustered, i.e. stored in the same texture, according to their cone of normals. This is the primary sorting criterion. Second, within one cone runs are grouped according to their spatial position, i.e. every texture is split into a set of smaller textures for which axis aligned bounding boxes are computed. This is the secondary sorting criterion. At run time, the CPU determines the partitions to be displayed based on the current viewing direction and the size and orientation of the view frustum. Only potentially visible partitions are send to the GPU, where they are finally decoded and rendered.

The CPU also determines the most appropriate LOD, i.e. grid resolution, to be rendered. We always select the resolution such that grid cells are always projected into an area smaller the size of one pixel under the current viewing parameters. An example of a hierarchical LOD representation with accompanied bounding box hierarchy is shown in Figure 11.

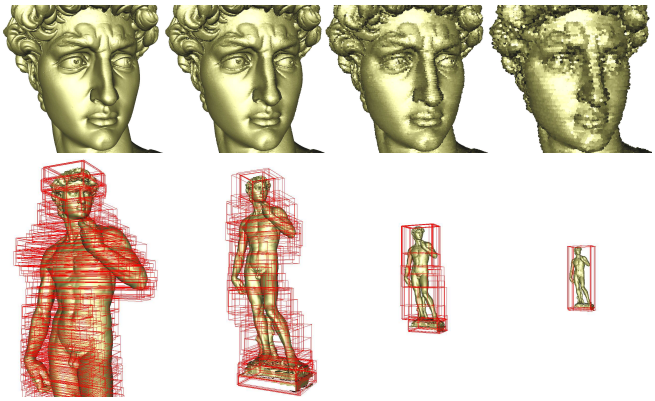

Fig. 11 The upper row shows closeups of David's head rendered at 4 different LOD levels. The lower row shows the corresponding image of the David statue as it would be rendered. The bounding boxes enclose parts of the mesh that are tested for frustum culling.

3 Results

The efficiency and effectiveness of the proposed point based rendering system were verified using the large scans from the Digital Michelangelo Project as well as a couple of iso-surfaces from volumetric data sets (see Figures 12, 13, 17). Above all, the richness in detail should be noted, which is faithfully reproduced at high fidelity using our approach. Table 1 shows comprehensive results for the largest meshes of the Digital Michelangelo

Project archive as well as for the iso-surfaces. Many of these data sets are excellent examples of surfaces where mesh simplification would not yield a dramatically reduced file size, in particular the fluid data set has the property that any given isovalue results in hundreds of millions of triangles.

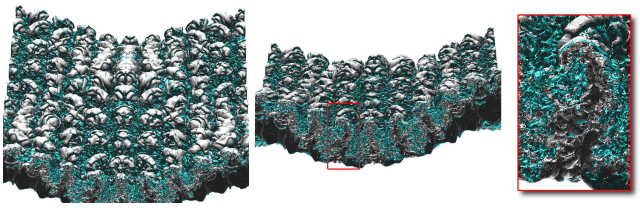

Fig. 12 Another time step of the mixing fluid simulation. In this image again two iso-surfaces (cyan and white))are rendered simultaneously. Note the fine scale detail visible on the bottom, where the mixing fluids form thousands of vortices.

Our target architecture is a P4 2.8 GHz CPU equipped with 1GB RAM. We timed our engine with an ATI Radeon X800 XT as well as an NVIDIA Geforce 6800 Ultra both with 256MB. The Radeon card used the render to vertex buffer functionality while on the Geforce the vertex texture fetch was used. Both systems achieved about the same performance while only with the Geforce the high quality rendering was possible due to the lack of floating point blending on the Radeon X800 card.

Fig. 13 The David and Atlas statues from the Digital Michelangelo Project.

In all point scan examples, run optimization resulted in a hit rate below 1.7 and a run efficiency above 70% of a maximum length of 25. As can be seen in Table 1, even for the atlas mesh the algorithm returns the result in less than 10 hours. Due to the hit rate larger than 1, the original point sets were reduced by a factor of 1.3 to 1.6.

In particular for the point scans it is obviously clear, that the point clustering approach as described introduces sampling errors. For the presented high resolution examples, these errors are 0.11mm and 0.14mm. The scanners used for the Digital Michelangelo Project have a minimum sample spacing of 0.25mm x 0.25mm x 0.1mm in a plane perpendicular to the laser [Cyb99]. In the worst case, two sample points are as much as

$$\sqrt{2 \times 0.25^2 + 0.1^2} mm \approx 0.367 mm$$

apart, which is the minimum size of features that can be faithfully reconstructed by the scanning process. Because in all our examples the sampling spacing is significantly higher than the sampling error introduced by our compression method, features present in the original data sets will not be destroyed. If the scanning device has sampled the data above the Nyquist rate of the original signal, our sampling is well above this rate, too, resulting in equal visual quality of the original and the compressed point set (see Figure 14).

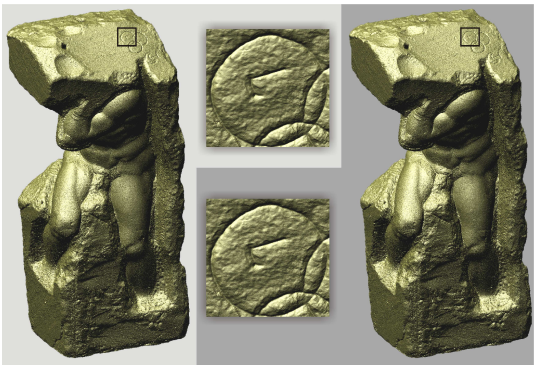

Fig. 14 Comparison of the original Atlas point scan [LPC*00] including normals (6 GB) to the point scan that was compressed by our method (231 MB). Note how the fine scale detail is preserved.

Table 1 also shows the excellent compression ratio our method achieves for real-world data sets exhibiting fine scale details. When using ZIP compression, the fluid interface iso-surface takes about 120 MB of space and while a single iso-surface in the Wholebody data set only occupies around 5 MB. The DuoDecim encoded VRIP versions of all of the Digital Michelangelo statues available on the web requires about 380MB and can thus be stored twice on an ordinary CD. Plain encoded without ZIP compression, it is still small enough to be stored in core of our target architecture. Due to the slice based encoding scheme for point sets, which is at the core of our technique, it is also well suited for streaming processing and progressive transmission of the data [IL04].

While the iso-surfaces and the point scans are considerably compressed, they can still be decoded very efficiently due to the simplicity of the decoding scheme. In the table we give timings for GPU decoding *and* rendering. To measure these timings, all acceleration techniques were all switched off, therefore these timings are considerably slower compared to the display times in practice. If decoding is carried out on the CPU, we observe a loss in performance of about a factor of 13.

Model	Atlas	St. Matthew	David
scan resolution	0.25 mm	0.25 mm	1.00 mm
# Points	254904158	186865425	28184522
# Samples	158877859	121718168	17190274
hit rate	1.60	1.53	1.64
run efficiency	72%	74%	72%
max sampling error	0.11 mm	0.14 mm	0.48 mm
ply file size	9.94 GB	7.29 GB	1.1 GB
DD compressed size	231 MB	182 MB	28.5 MB
zip compressed file	172 MB	140 MB	21 MB
DD encoding time	9.5 hrs	6 hrs	57 min.
decode & render time	3.91 sec.	2.85 sec.	0.39 sec.

Model	Wholebody	Fluid
volume resolution	512 x 512 x 3172	2048 x 2048 x 1920
volume size	1.6 GB (short)	7 GB (byte)
triangulated mesh size	1.5 GB	24 GB
run efficiency	89%	81%
DD compressed size	20 MB	210 MB
zip compressed file	11 MB	120 MB
DD encoding time	1.7 hrs.	8.2 hrs
decode & render time	0.23 sec.	3.5 sec.

Table 1 Timing and memory statistics for the proposed point based rendering system. Timings where made on the ATI X800 XT card using DirectX.

Fig. 15 Zoom in on the Michelangelo's St. Matthew Statue, note the fine scale details even in the lower rightmost closeup.

On the GPU, the point rendering system achieves a throughput of about 50 million points per second. This rate includes the decoding of compressed point runs as well as the rendering of decoded points and normals. It is worth noting, that we decode and render about 160M distinct points and normals in roughly 3 seconds on the GPU. Figures 15 and 16 show some more examples that demonstrate the need for a point based rendering system able to handle such an amount of primitives. In all images, the point splat size is automatically set according to the screen space projection of the underlying grid cells.

4 Runtime & Memory Requirements

4.1 Iso-Surface Extraction

The runtime for iso-surface extraction heavily depends on the input grid type. For structured grids, we assume a HCP grid of resolution $N := v^3$. Clearly, interpolation and resampling is in $\mathcal{O}(N)$, as long as the filter size is constant. Since we stream the data set, we only need to store

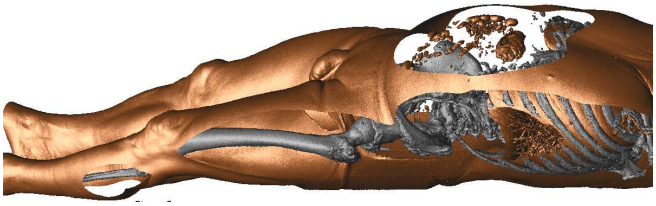

Fig. 16 Two iso-surfaces of a CT scan of a human male. The two surfaces consist of about 30 million points. The Iso-surface was compressed from about 1 GB to only 21 MB by our method. Data set courtesy of Siemens Corporate Research, Inc. Princeton

$2\rho+1$ slices at a time, where each slice contains $\mathcal{O}(N^{2/3})$ voxels. In the case of unstructured grids we usually have to perform a logarithmic cell search during interpolation, and hence the runtime is in $\mathcal{O}(N \log N)$. Memory requirements are still in $\mathcal{O}(N^{2/3})$.

In case the iso-surface should be changed after resampling, we store the entire HCP grid on disk to avoid repeated resampling.

4.2 Laser Range Scans Point Clustering

First, points are inserted into the HCP grid, which requires linear runtime. However, range scans are first sorted, which clearly is in runtime of $\mathcal{O}(N \log N)$. As it can be safely assumed that the surface of the object is dominated by 2-manifold topology, the ratio of filled voxels is $v^2{:}v^3$, where $N := v^2$ denotes the number of input points. Because we only hash filled voxels of one slice at a time, memory requirements are as low as $\mathcal{O}(\sqrt{N})$.

It is worth noting that many objects have a distinct major axis, and consequently slicing along this axis reduces memory requirements considerably.

4.3 Run Generation

So far a single slice of the data sets presented in this paper always fitted into core memory. Runs are then generated using our linear-time 2-approximation. However, if in the future data sets should be so large that a single slice will not fit into memory any more, we can still brick the data and process each brick independently. This does not affect the runtime of $\mathcal{O}(N)$ for the entire conversion.

4.4 Quantization

To quantize the positions in the run, we can traverse each run linearly. For the normals we first generate a codebook using vector quantization, which, if carefully implemented, is in $\mathcal{O}(\log(k) \cdot p)$, where k is the number of entries in the codebook and p is the number of vectors. However, since it is sufficient for large models to pick a small, representative subset p of constant size, obtaining

a normal codebook is negligible in the entire process. Memory requirements are $\mathcal{O}(k+p)$. Once the codebook is obtained, we assign each normal the closest entry. Since k in $\mathcal{O}(1)$, this is in $\mathcal{O}(N)$.

5 Conclusion

In this paper, we have presented an effective compression scheme for large iso-surfaces in any kind of volumetric data and for gigantic points scans. Our compression scheme is based on close sphere packing grids. Such grids provide a structure for optimal point clustering, and they establish a spatial relation between points that can be exploited for compression purposes. As our results have shown, the compression scheme achieves an extraordinary compression ratio at very high fidelity. Due to the simplicity of the decoding scheme, point coordinates and normals can be reconstructed on the GPU. As the GPU can also render the decoded primitives without any readback to the CPU, bandwidth requirements are substantially reduced.

In the future, we will extend the rendering engine about more elaborate space partitioning strategies, which allow for improved culling if the user zooms into the data set or if only a small portion of the data is rendered.

6 Acknowledgements

We would like to thank Siemens Corporate Research, Inc. Princeton for the Wholebody CT scan, Peter Lindstrom for the mixing interface simulation data set, and the people from the Digital Michelangelo Project for scanning the statues and making the mesh data public.

References

[ABCO*01] Alexa M., Behr J., Cohen-Or D., Fleishman S., Levin D., Silva C. T.: Point set surfaces. In *Proceedings of Visualization '01* (2001), pp. 21–28.

[AG92] Allen Gersho R. M. G.: *Vector Quantization and Signal Compression*. Kluwer International Series in Engineering and Computer Science, 1992.

[ATI04] ATI: Superbuffers OpenGL Extension. www.ati.com/developer/gdc/SuperBuffers.pdf, 2004.

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

[BK03] Botsch M., Kobbelt L.: High-quality point-based rendering on modern gpus. In *Proceedings of Pacific Graphics 2003* (2003), p. 335.

[BWK02] Botsch M., Wiratanaya A., Kobbelt L.: Efficient high quality rendering of point sampled geometry. In *Proceedings of the 13th Eurographics workshop on Rendering* (2002), pp. 53–64.

[CAZ01] Cohen J. D., Aliaga D. G., Zhang W.: Hybrid simplification: combining multi-resolution polygon and point rendering. In *VIS '01: Proceedings of the conference on Visualization '01* (2001), pp. 37–44.

[CN01] Chen B., Nguye M. X.: Pop: A hybrid point and polygon rendering system for large data. In *VIS '01: Proceedings of the conference on Visualization '01* (2001).

[CSB87] Conway J. H., Sloane N. J. A., Bannai E.: *Sphere-packings, Lattices, and Groups*. Springer-Verlag New York, Inc., 1987.

[Cyb99] Cyberware: 3D Scanner Designed To Scrutinize Works Of Michelangelo. www.cyberware.com/news/pressReleases, 1999.

[DH02] Dey T. K., Hudson J.: Pmr: point to mesh rendering, a feature-based approach. In *VIS '02: Proceedings of the conference on Visualization '02* (2002), pp. 155–162.

[DVS03] Dachsbacher C., Vogelgsang C., Stamminger M.: Sequential point trees. *ACM Computer Graphics (Proc. SIGGRAPH '03) 22*, 3 (2003), 657–662.

[GD98] Grossmann J., Dally W.: Point sampled rendering. In *Proceedings Eurographics Rendering Workshop* (1998), pp. 181–192.

[GM04] Gobbetti E., Marton F.: Layered point clouds. In *Eurographics Symposium on Point Based Graphics* (2004), pp. 113–120, 227.

[GM05] Gobbetti E., Marton F.: Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Trans. Graph. 24*, 3 (2005), 878–885.

[GPA*] Gross M., Pfister H., Alexa M., Pauly M., Stamminger M., Zwicker M.: Point-based computer graphics. SIGGRAPH '04 Course Note.

[IBM] IBM Corp. T221 Flat Panel Monitor. http://www.ibm.com.

[IL04] Isenburg M., Lindstrom P.: *Streaming meshes*. Technical Report UCRL-CONF-201992, LLNL, 2004.

[Joh66] Johnson N. W.: Convex Polyhedra with Regular Faces. *Canadian Journal of Mathematics 18* (1966), 169–200.

[KB04] Kobbelt L., Botsch M.: A survey of point-based techniques in computer graphics. *Computers & Graphics 28*, 6 (2004), 801–814.

[KSW05] Krüger J., Schneider J., Westermann R.: Duodecim - a structure for point scan compression and rendering. In *Proceedings of the Symposium on Point-Based Graphics 2005* (2005).

[LC87] Lorensen W., Cline H.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Computer Graphics (SIGGRAPH 87 Proceedings)* (1987), pp. 163–169.

[Lev90] Levoy M.: Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics 9*, 3 (July 1990), 245–261.

[Lev00] Levoy M.: Digitizing the forma urbis romae. In *Proceedings of the ACM SIGGRAPH and Eurographics Campfire Workshop on on Computers and Archeology* (Snowbird, Utah, USA, 2000).

[LH98] Livnat Y., Hansen C.: View dependent isosurface extraction. In *VIS '98: Proceedings of the conference on Visualization '98* (Los Alamitos, CA, USA, 1998), IEEE Computer Society Press, pp. 175–180.

[LPC*00] Levoy M., Pulli K., Curless B., Rusinkiewicz S., Koller D., Pereira

L., Ginzton M., Anderson S., Davis J., Ginsberg J., Shade J., Fulk D.: The digital michelangelo project: 3D scanning of large statues. In *ACM Computer Graphics (Proc. SIGGRAPH '00)* (2000), pp. 131–144.

[LSJ96] Livnat Y., Shen H.-W., Johnson C. R.: A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics 2*, 1 (1996), 73–84.

[LW85] Levoy M., Whitted T.: *The use of points as a display primitive.* Technical Report 85-022, University of North Carolina at Chapel Hill, 1985.

[Mat04] Mattausch O.: Practical reconstruction and hardware-accelerated direct volume rendering on body-centered cubic grids. In *CESCG 2004* (2004).

[MH01] Mroz L., Hauser H.: Space-efficient boundary representation of volumetric objects. In *Proceedings IEEE/TVCG Symposium on Visualization 2001* (2001), pp. 180–188.

[Mic04] Microsoft: Shader Model 3 Specification. http://msdn.microsoft.com, 2004.

[NM02] Neophytou N., Mueller K.: Space-time points 4d splattin on efficient grids. In *IEEE Symposium on Volume Visualization '02* (2002), pp. 97–106.

[OS04] Ochotta T., Saupe D.: Compression of point-based 3d models by shape-adaptive wavelet coding of multi-height fields. In *Eurographics Symposium on Point Based Graphics* (2004).

[PZvBG00] Pfister H., Zwicker M., van Baar J., Gross M.: Surfels: surface elements as rendering primitives. In *ACM Computer Graphics (Proc. SIGGRAPH '00)* (2000), pp. 335–342.

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

[RPZ02] Ren L., Pfister H., Zwicker M.: Object space ewa splatting: A hardware accelerated approach to high quality point rendering. In *Proceedings Eurographics 2002* (2002), pp. 371–378.

[SK01] Saupe D., Kuska J.: Compression of isosurfaces. In *Proceedings of Vision, Modeling and Visualization '01* (2001), pp. 330–340.

[WE98] Westermann R., Ertl T.: Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics (SIGGRAPH 98 Proceedings)* (1998), pp. 291–294.

[WWL*04] Waschbüsch M., Würmlin S., Lamboray E. C., Eberhard F., Gross M.: Progressive compression of point-sampled models. In *Eurographics Symposium on Point Based Graphics* (2004).

[ZPvBG01] Zwicker M., Pfister H., van Baar J., Gross M.: Surface splatting. In *ACM Computer Graphics (Proc. SIGGRAPH '01)* (2001), pp. 371–378.

[ZRB*04] Zwicker M., Rasanen J. J., Botsch M., Dachsbacher C., Pauly M.: Perspective accurate splatting. In *Proceedings of Graphics Interface 2004* (2004), pp. 247–254.

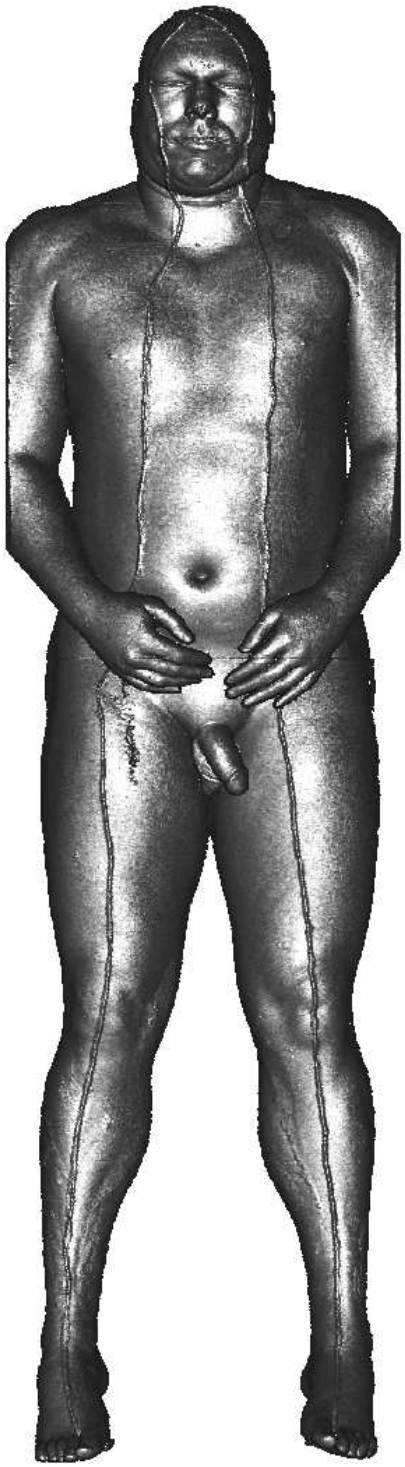

Fig. 17 An isosurface of the Visible Human male dataset. While the volume data requires about 1 GB the compressed data stream rendered in this image is only about 5MB in size.