Eurographics Symposium on Point-Based Graphics (2007)
M. Botsch, R. Pajarola (Editors)

A Parallelly Decodeable Compression Scheme for Efficient Point-Cloud Rendering

Ruwen Schnabel Sebastian Möser Reinhard Klein†

Computer Graphics Group, University of Bonn, Germany

Abstract

We present a point-cloud compression algorithm that allows fast parallel decompression on the GPU for interactive applications. We achieve bitrates of less than four bits per normal-equipped point. Our method enables hole-free level-of-detail point rendering. We also show that using only up to two bits per point, high-quality renderings can still be obtained if normals are estimated in image-space. The algorithm is based on vector quantization of an atlas of height-fields that have been sampled over primitive shapes which approximate the geometry.

Categories and Subject Descriptors (according to ACM CCS): E.4 [Coding and information Theory]: Data compaction and compression I.3.3 [Computer Graphics]: Display algorithms I.3.3 [Computer Graphics]: Digitizing and scanning I.3.3 [Computer Graphics]: Bitmap and framebuffer operations

1. Introduction

The development in 3D laser scanning technology has led to relatively low-cost devices capable of easily capturing very large scenes, such as buildings, streets, historical artifacts or industrial compounds. With the increasing availability of these devices even to small and medium sized businesses grows the demand for efficient algorithms to handle the huge datasets generated by the scanners. Let alone reconstruction and visualization, even seemingly simple tasks such as storage itself often pose a challenge for these gigantic models. More importantly though, in spite of increasing bandwidth, it is still difficult to share these large datasets across a network. We believe that in this context high compression rates are of major importance since we expect the size of datasets to grow much faster than e.g. hard-disk transfer rates in the future, while the increased computational demand of decompression is met well by the continuous and impressive gains in GPU processing performance. It is therefore desirable to have a format for this data that achieves very high compression rates while lending itself to quick decompression and rendering at the same time.

In this paper we present a novel algorithm that achieves just that: Point-cloud data is decompressed parallelly on the GPU and is rendered at interactive rates. For large models the compression rates of our scheme are similar to state-of-the art sequential point-cloud compressors. The main features of our algorithms are:

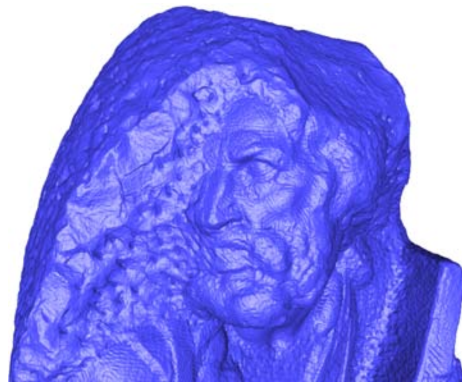

Figure 1: *Michelangelo's St. Matthew rendered interactively at 3.31bpp including normals.*

Bitrate We show that interactive high quality rendering at about 5-10fps on current hardware is achieved with *less than four bits* per input point including normals (see Fig. 1).

† e-mail:{schnabel,moeser,rk}@cs.uni-bonn.de

© The Eurographics Association 2007.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

Normal estimation Our system allows trading compression of normals for image-space normal estimation. This way coding of point positions alone suffices to obtain realistic renderings and point-clouds can be interactively rendered from *less than two bits* per point.

Level-of-detail Our method uses progressive compression and inherently supports level-of-detail.

In our method, the point-cloud is decomposed into patches that are each approximated by a primitive shape, i.e. either a plane, sphere, cylinder, cone or torus. The fine-scale geometry is then encoded as height-fields over these patches. These height-fields are compressed progressively using image-based methods.

The shapes provide a close approximation of the geometry for arbitrary models, but the shape information can also be used to allow some primitive user interaction, such as suppressing the rendering of selected items, moving or copying them. In scenes such as buildings, cities or other man-made environments these shapes are predominant and often are closely related to semantic entities. Moreover, in these scenes, the overall compression rate can be higher since the geometry is even more closely approximated by shape primitives.

2. Previous work

One of the first approaches combining compression with direct rendering was the QSplat system [RL00] which is based on a hierarchy of bounding spheres, giving a level-of-detail representation. The positions and radii (as well as other attributes) are delta coded in the hierarchy to reduce memory consumption. They require 6 bytes per input point with normal.

Botsch et al. [BWK02] use an octree as hierarchy for compression as well as rendering and sample the characteristic function of the surface into this representation. To encode the hierarchy in a coarse to fine manner, only the subdivisions of non-empty cells have to be stored in byte codes. This way they require less than 2 bits per leaf node. The number of leaf nodes is directly linked to the resolution of the finest octree level, since the characteristic function is continuous. Therefore, in general, more leaf nodes than original points are required to represent a model at its full precision, leading to much higher bitrates. To reduce the required grid precision, in each leaf, they store small offsets in normal direction, quantized to additional 2 bits.

Kalaiah and Varshney [KV05] use a statistical representation of the point-cloud to define a level-of-detail hierarchy. The hierarchy is constructed by computing a PCA of the point positions for each node and than dividing along the most significant axis. The parameters of the PCA, i.e. the local frame and the variances, are quantized to 13 bytes per node. Since leaf nodes represent a cluster of points, the hierarchy requires only about 8-9 bits per input point. The rendering of a node is based on quasi-random sampling of the encoded Gaussian distribution. By precomputing a sequence of quasi-random numbers for a unit Gaussian distribution the sampling can be shifted to the vertex shaders of the GPU. The decoding of the node parameters however has to be done by the CPU.

Krueger et al. presented DuoDecim [KSW05], a point-cloud compression algorithm suitable for real-time GPU decompression. They resample the original point-cloud into a grid composed of Trapezo Rhombic Dodecahedra (TRD). Since a cell in a grid of TRDs has no second order neighbors, adjacency relations between cells can be encoded very effectively in only 2.25 bits. Thus, for compression of the grid, continuous runs of neighboring occupied cells are stored based on the simple adjacency relations. For decompression, several of these runs can then be processed on the GPU in parallel. The method achieves high compression rates of about 3bpp for positions and 5bpp for normals while introducing only a small sampling error. However, several grids have to be stored independently to obtain a level-of-detail representation.

Compression of point-sampled geometry without having direct rendering in mind has also been studied extensively. Most related to our approach is the work of Ochotta et al. [OS04]. They partition the point-cloud into patches parameterizable over a plane. Similar to us, they resample the geometry as height-fields over these planes. Then they use progressive wavelet image compression on these height-fields to achieve bitrates of 2-3bpp for point positions. Other approaches in this area include the tree-based methods given in [HPKG06] [SK06] and the algorithm provided by [WGE*04]. While all these methods are progressive and achieve high compression rates, in contrast to our approach, decompression is sequential and cannot be performed on the GPU for interactive rendering. Nonetheless our algorithm is able to achieve similar bitrates on large models.

The height-field geometry representation employed in our system is a well known concept that, besides its use in compression (see above), spectral analysis [PG01], simplification and reconstruction [BHGS06], has also been used for rendering. Ivanov and Kuzmin [IK01] propose the use of planar range images as rendering primitive in a hardware pipeline. However, they use a large number of very small patches and do not consider any compression. Ochotta and Hiller [OH06] designed a rendering system based on height-fields that achieves high quality renderings at interactive rates. However, they too, do not consider compression or level-of-detail.

3. Overview

Our method is an asymmetric compression algorithm that is based on vector quantization of the height-fields' Laplace

© The Eurographics Association 2007.

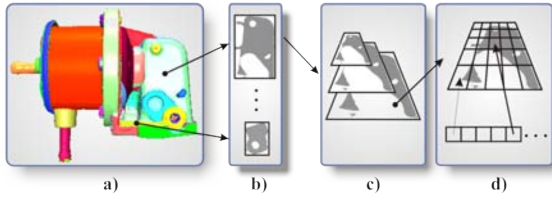

Figure 2: *Different stages in our compression algorithm. a) The object is decomposed into parts corresponding to shape primitives. b) Height fields over the primitives are generated to describe fine scale details. c) Laplace pyramids are computed for each height field d) Pyramid levels are encoded with vector quantization.*

pyramids. The first step is the decomposition of the input point-cloud into parts parameterizable over a primitive shape (depicted in Fig. 2 a)). To this end we employ a recently proposed efficient RANSAC shape detection scheme that we have extended with regard to selecting shapes more suitable for compression (see sec. 4).

Once the decomposition has been obtained, the geometry will be represented as an atlas of height-fields that have been resampled on a regular grid located in the domain of the respective primitive [PG01] [OS04] (Fig. 2 b)). Note that these height-fields are allowed to have irregular boundaries and we store with each field a binary occupancy mask to encode the existence of surface samples in the corresponding grid cells. Since height-fields are basically equivalent to grey-scale images we also refer to the height-fields as images throughout this work.

Each height-field is filtered and downsampled to yield a collection of equally high image-pyramids (Fig. 2 c)). Starting with the coarsest pyramid level, the images of all shapes are simultaneously vector quantized. The quantized versions are then upsampled and subtracted from the next finer resolution images, resulting in difference images. Such difference images are then successively vector quantized for each pyramid level (see Fig. 2 d) and sec. 5).

Decompression then simply replaces codebook indices with codebook vectors and sums up as many difference images as required to reconstruct a selected pyramid level. This can be done efficiently on the GPU using dependent texture lookups (see sec. 6). The resulting height-fields are then reconverted into point-clouds for rendering.

During rendering, level-of-detail is realized by choosing a pyramid level for each patch such that the level's resolution guarantees a hole-free point rendering. Should a hole-free rendering require a higher resolution than available, a lower resolution framebuffer is chosen as render target. These lower resolution images are later scaled and merged into the target resolution to obtain the final rendering. During this image processing, normals can also be generated from the stored depth information (see sec. 7).

4. Decomposition

We decompose the input data using the robust RANSAC-based approach that was suggested in [SWK07]. Here we will only shortly review this algorithm and then present the extensions we implemented with regard to compression.

The point-cloud $P = \{p_1, \ldots, p_N\}$ is decomposed into subsets S_i associated to shape primitives Φ_i as well as a single subset R containing any remaining points that could not be assigned to a shape. Thus, after decomposition $P = S_1 \cup \ldots \cup S_A \cup R$. In the following we will use the terms shape and primitive interchangeably.

RANSAC shape detection is a probabilistic algorithm that randomly generates shape hypotheses. These hypotheses are tested against the point-cloud by evaluating a score function σ. Primitives achieving maximal score are extracted iteratively from the point-cloud in a greedy manner. Each time a primitive has been extracted, all compatible points forming a connected component on the surface are removed from P and collected in a subset S_i. In order to heuristically ensure that only parameterizable patches are created, a point is considered compatible if it is within a distance of ε and its normal does not deviate by more than α degrees from the respective shape normal. After removing the compatible points, the algorithm is restarted on the remaining points until no more shapes can be found.

The output of the method can easily be adjusted to special application requirements by defining a suitable score function σ. In our case we wish to extract shapes that, besides approximating a large number of points, should also have a short boundary in the shape's parametrization domain. The motivation behind this is that the boundary will have to be encoded by the compression later on, and more complex boundaries will lead to higher encoding cost. Thus, we define the score function as

$$\sigma(P, \Phi) = |S(P, \Phi)| - \beta |B(S(P, \Phi))|,$$

where Φ is the primitive, $S(P, \Phi)$ is the largest set of compatible points forming a connected component and $B(S(P, \Phi))$ is the set of boundary points in the primitive's domain. The parameter $\beta \leq 1$ allows adjusting the influence of the boundary points.

The decompositions sorts outliers and points belonging to very scarcely sampled regions into the set of remaining points R. These points can safely be ignored in all further processing.

5. Compression

After the shape primitives have been obtained, the height-field atlas has to be generated. To this end, the point-cloud

© The Eurographics Association 2007.

is resampled on a regular grid in the domain of each shape. Recall that the compressed point-cloud will consist only of these resampled positions, so that care has to be taken to avoid visible gaps between the edges of different shape primitives. As was observed by Ochotta et al. in [OH06], in order to obtain hole free rendering, it suffices to have the patches slightly overlap.

5.1. Resampling

The height-field patches are regularly sampled in the parametrization domain of the respective shape primitive. To establish the location and extent of the resampling grid for each patch, all points assigned to a shape are projected onto the respective primitive and a bounding box is found for all these projected points in the parametrization domain. In order to preserve the original number of points, the resolution of each resampling grid should be chosen such that the number of occupied cells equals the number of projected points of the respective patch. The result of the resampling is a regularly sampled height-field or grey-scale image together with a binary mask specifying the valid image entries.

We find the height-field's resolution and the binary mask of occupied cells in a joint iterative process. The initial resolution of the grid is set to the average distance between a projected point and its nearest neighbor. In each iteration the projected points are sorted into the grid and a morphological closing operation is used to fill small holes in the resulting occupancy mask. Then, similar to a binary search, if the number of occupied cells after the closing is larger than the original number of points, the resolution is set to the middle value between a lower bound and the current resolution. Otherwise the resolution is increased analogously. After the correct resolution has been determined, the constructed binary mask is dilated once to ensure a slight overlap between neighboring patches in space.

Now, for each occupied cell a height value has to be computed. These height values are obtained by intersecting a ray in direction of the shape primitive's normal with the moving least-squares (MLS) surface for each masked cell [AA03]. Using the MLS surface has the advantage that different patches use a consistent surface definition across patch borders, which ensures that no edges will be visible in the resampled point-cloud. In addition, the MLS surface can also be used to obtain normals, which can be stored along with the offset value if desired. Point normals are encoded relative to the primitive's normal using spherical coordinates. Using the primitive's normal as reference results in a low entropy for the polar angle, as it will usually be close to zero.

5.2. Filtering

The acquired height-fields are successively filtered and subsampled to obtain image pyramids. The levels of these pyramids constitute the levels-of-detail supported by our method. A level P_i of the pyramid is obtained as $P_i = \downarrow g(P_{i-1})$, where g is a low-pass analysis filter and $\downarrow$ denotes subsampling. P_0 is the original image. Rather than storing the pyramid levels independently we use the Laplacian pyramid representation introduced by Adelson and Burt [AB81] to achieve better decorrelation (and thus compression). A level of the Laplacian pyramid is the difference between the corresponding level of the image pyramid and the upscaled lower resolution level. Thus a level L_i of the Laplacian pyramid is given as $L_i = P_i - h(\uparrow P_{i+1})$, where h is a synthesis filter and $\uparrow$ denotes upsampling. Only on the coarsest level l, Laplacian and image pyramid are identical, i.e. $L_l = P_l$. We can then reconstruct a level P_i from the Laplacian pyramid by recursively applying $P_i = L_i + h(\uparrow P_{i+1})$.

Originally, Adelson and Burt suggested to use Gaussian-like filter kernels for g and h in the pyramid construction. However, Gaussian analysis filters also require Gaussian synthesis during reconstruction of pyramid levels. With a GPU implementation of the reconstruction in mind, we use the CDF 5/3 [CDF92] wavelet instead, as in this case the low-pass synthesis filter simply is a bilinear interpolation, which is natively supported in the hardware.

5.3. Vector quantization

Vector quantization works by replacing small tiles of the original image with indices into a codebook [GG92]. The codebook simply contains a set of representative tiles. The main reason to use vector quantization in our method is that this simple scheme directly lends itself to parallel decompression while achieving high compression ratios. To obtain the decompressed image all indices can be replaced with the vectors from the codebook independently and concurrently. In principle, all this amounts to are dependent texture lookups on the GPU.

Several related approaches have also used vector quantization to enable fast decompression. Beers et al. [BAC96] introduced the concept to the computer graphics community in the context of texture compression and Levoy et al. used vector quantization for Light field rendering [LH96]. In [SW03] Schneider et al. showed that the scheme can also be applied to compression and rendering of volume data.

Thus, to achieve compression we apply vector quantization to the pyramid levels L_i [HH88]. For the vector quantization the height-fields are decomposed into small vectors corresponding to square tiles of side length x. Each vector carries a mask, that has been generated from the occupancy bitmap, to identify any missing values. No vectors are generated for empty regions in the patch.

The key to high compression rates is to find a small codebook together with a mapping from original tiles to codevectors such that the distortion introduced by the replacement is minimized. Let $\mathbf{C} = \{c_i, \ldots, c_k\}$ be a set of codevectors and $\mathbf{D} = \{d_i, \ldots, d_N\}$ be the set of data-vectors (or

© The Eurographics Association 2007.

image tiles respectively), then distortion is measured as the root mean square (RMS) error:

$$R = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \|d_i - c_{M(i)}\|^2},$$

where $M : \mathbb{N} \rightarrow \mathbb{N}$ is the mapping assigning code-vectors to data-vectors.

In our system the user specifies a maximal RMS error R_{max} prior to compression, such that a codebook and mapping have to be found accordingly. While we treat different pyramid levels independently, on each level we use a common codebook across all shape patches, i.e. the data-vectors **D** are collected on the respective level from all patches in the atlas. We do not quantize across scales for two reasons: Firstly, we can avoid accumulating quantization errors if we compute the levels of the Laplace-pyramid using the previously quantized lower resolution images. Secondly any codebook across scales has to be large enough to achieve an error less than R_{max} on L_0. This leads to an overly verbose codebook on coarser levels and, consequently, to large code-vector indices requiring many bits.

5.3.1. Codebook generation

We base the codebook generation on the LBG algorithm [LBG80]. It is well known however that a naive implementation of this method exhibits extremely poor runtime performance. We employ simple strategies to alleviate this problem without notably degrading compression performance: Firstly, we quickly obtain an initial codebook using a tree structured vector quantization (TSVQ) approach [GG92]. Secondly, the codebook of the TSVQ is refined with the generalized Lloyd algorithm. The Lloyd iterations are accelerated through approximate nearest neighbor computations.

TSVQ The TSVQ proceeds hierarchically, as described e.g. in [LGK*01] [SW03] [KV05]. Since the codebook will be further optimized after the TSVQ, we stop adding code-vectors once a distortion less than γR_{max} has been achieved. We empirically found $\gamma = 1.5$ to be a good choice.

Lloyd The codebook as well as the mapping created by the TSVQ are far from optimal. Therefore it is worthwhile to refine the code vectors with subsequent iterations of the generalized Lloyd algorithm. The Lloyd algorithm finds an optimal mapping M by assigning each data-vector to its nearest code-vector. Optimal code-vectors are then computed as the centroid of all data-vectors which they are to represent. This is repeated until convergence. Even though the Lloyd algorithm significantly reduces the RMS error that was achieved by the TSVQ, it usually is unable to reach R_{max} without adding additional code-vectors. Therefore we interleave Lloyd and TSVQ until R_{max} is reached. In each iteration TSVQ is used to generate additional code-vectors in relevant locations [SW03].

Approximate Lloyd Finding the nearest code-vectors in each iteration of the Lloyd algorithm is a very expensive operation, which is why we resort to an approximation instead: We find the list of m nearest code-vectors for each data-vector in the first iteration and then restrict the search for the nearest code-vector to this list in all subsequent iterations. This way the search is significantly accelerated and the runtime is indeed now dominated by the first iteration. However, a small error is introduced, since code-vectors may change in between iterations such that at some point the list may no longer contain the true nearest neighbor.

Scalar quantization After the codebook generation, the elements of the code-vectors additionally undergo scalar quantization into eight bits per element. This way they can be stored in single component textures on the GPU.

5.4. Hierarchy

In principle, it is possible to store the compressed pyramid levels as two dimensional arrays of code-vector indices. However, this would imply saving indices even in empty regions of a level. Since the occupancy masks may indeed contain large empty areas, this wastes a lot of space with useless information. Thus it is significantly more efficient to use a quad-tree representation of the pyramid, in which only the occupied areas have to be stored.

5.4.1. Quad-tree

Traditionally, each node of the quad-tree would contain a code-vector index together with a list of pointers to the existing child nodes. The list of pointers can be replaced by four bits specifying the existing children if the quad-tree nodes are stored in breadth- or depth-first order. However, during decompression we want to be able to process many quad-tree nodes in parallel on the GPU, and while such a representation is very space efficient, it is not well suited for parallel processing due to the sequential nature of the traversal order. Thus, in the spirit of the well-known recursive data pattern [MSM04], to enable efficient parallel decompression we store instead a pointer to its parent together with two bits specifying its child relation. We keep the levels of the quad-tree in separate arrays, such that the pointers actually are offsets into these arrays. This representation allows us to process each node on a level in parallel with all other nodes of the same level at the cost of additional pointers in the leaf nodes.

Parent pointers The encoding of parent indexes in the quad-tree nodes may become problematic if the images are very large, and therefore indices into large arrays would have to be stored. Thus, to avoid spending to many bits on the parent indices, we subdivide every image into square blocks of side length $2^l x$ (recall that l is the number of pyramid levels and x is the side length of the quantized image tiles). This

© The Eurographics Association 2007.

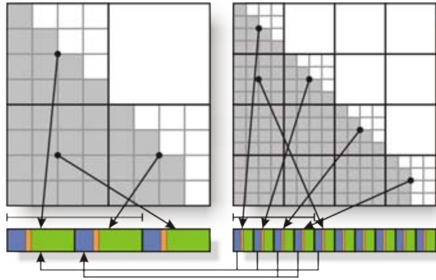

Figure 3: *Two consecutive levels of an image quad-tree. Each quad-tree cell contains x^2 pixels. Below the quad-tree tiles the array for the level is depicted. Each entry stores a pointer to the parent tile, the child relation and the code-vector index. Array entries corresponding to partial tiles, i.e. tiles with incomplete occupancy masks, are sorted to the beginning of the array.*

limits the number of bits required to store parent offsets to $2(l-1)$ on the finest level. Moreover, as a side effect, we can use these sub-blocks to define the granularity of our level-of-detail selection. To this end, we also store a bounding box along with each block.

For a compression using five levels and using a tile side length of four the expected bits per point required on the last level for the parent indices can be now be computed as $\frac{2(l-1)}{x^2} = \frac{1}{2}$. This is still a significant amount which we can further reduce by sorting the tiles of the last and next to last level such that nodes that have four children are stored in an order which allows implicit quad-tree indexing of the parent (and therefore of the child relation as well). This way we usually save more than 50% of the overhead caused by parent indexes on the last level.

Occupancy bitmaps The number of bits needed for a node of the quad-tree now depends on the number of bits required to encode a code-vector index, the offset into the parent array and two bits for the child relation. Unfortunately however, it does not suffice to store the code-vector index alone for decompression, as some of the quad-tree cells may only be partially occupied. For these cells a bitmap is used to encode the occupancy. Please note that we call a quad-tree node partial if not all of its associated image pixels are occupied, which is independent of the number of children of the node.

To minimize storage overhead for the occupancy bitmaps, all partial quad-tree tiles of each level are sorted to the beginning of the level's array and the corresponding bitmaps are stored in the same order in a second array. Fig. 3 illustrates the resulting layout of the data structure. This way, the only overhead is the number of partial nodes that has to be encoded for each level and no additional information is needed for full nodes.

6. Decompression

The aim of our compression technique is to allow for fast decompression on the GPU, which has two advantages: Firstly only the compressed data has to be sent across the bus to the GPU during rendering and secondly the high degree of parallelism of the GPU's SIMD structure can be fully exploited.

The decompression of a patch reconstructs the quad-tree level corresponding to the level of the image pyramid which has been selected for rendering. As the result of the decompression the reconstructed quad-tree tiles will be stored in a vertex buffer. The vertex buffer contains four floating point values per reconstructed point or six if normals are also decompressed. These values are the height value h, two coordinates u and v specifying the location of the point in the primitive's domain, as well as a value b that is zero if the point corresponds to a position that was masked out by the occupancy bitmap. In case of decompressed normals there are two additional values ϕ and θ. Since each quad-tree node specifies an index of a single code-vector, a node decompresses into exactly x^2 points. Using (u,v) the height and normal values of the points will be transformed into world-coordinates with respect to the shape primitive in a vertex shader during rendering. Points with $b = 0$ are discarded in a geometry shader.

Thus, for reconstruction of a level P_j, the arrays of the quad-tree for level j are uploaded into textures on the GPU. The reconstructed nodes of the previous level P_{j+1} are copied into textures as well. If the number of nodes in the quad-tree on level j is k, than kx^2 points have to be reconstructed, since every node encodes a x^2 image tile. We use the new transform feedback OpenGL extension, render kx^2 points and perform the decompression of each point in a vertex shader. The transform feedback stores the result of the vertex shader directly into a vertex buffer which can then directly be used for rendering without any prior copying.

The decompression vertex shader reads the respective node information from the quad-tree array. Note that to achieve maximal concurrency the node information is actually read many times - once for each point. Due to caching of the data this causes virtually no overhead however. All that has to be done for reconstruction is to add the code-vector entry, which is read from a codebook texture, to the respective interpolated parent value. The interpolation is handled automatically in the texture unit. Additionally the point's coordinates are deferred from the parent coordinates. Note that the interpolation of parent values is restricted to the values belonging to the parent node by adjusting the texture coordinates accordingly. This causes a slightly decreased compression performance but offers the advantage that no neighboring quad-tree nodes have to be considered in the decompression. Thus, computations as well as data structures are significantly simplified.

© The Eurographics Association 2007.

7. Rendering

In principle, during rendering the decompressed height values only have to be transformed into 3D coordinates in a vertex shader and can than be rastered as simple point primitives. However we want to incorporate level-of-detail control for better performance. Also hole-free renderings of close-up views is desired.

7.1. Level-of-detail

With our system, level-of-detail control can be achieved fairly simply: For each patch the distance of the bounding box to the viewer d is obtained. Since the sampling resolution of the patch is known this distance can be used to select the level-of-detail as follows:

$$l_{LOD} = -\log_2\left(\frac{\sqrt{2}nr_{patch}}{dp}\right),$$

where r_{patch} is the resolution of the patch, n is the distance to the near plane and p is the side length of a pixel in world coordinates. This choice guarantees a hole-free rendering of objects as long as $l_{LOD} \geq 0$. For patches with $l_{LOD} < 0$ we use a hierarchy of coarser framebuffers to obtain hole-free renderings. The levels of this framebuffer hierarchy are merged into a single image for each frame.

7.2. Hole-free rendering

To achieve a hole-free point rendering, splatting approaches such as those proposed in [ZPvBG01] or [BK03] usually are a first choice. Splatting however requires a normal for each surface element so that it cannot be directly applied if normals have to be estimated in image-space. Also splatting requires the geometry to be rendered in two passes, which causes significant overhead for large models. The pyramid of framebuffer images that we employ instead requires only a single geometry rendering pass followed by a few very fast image-based passes. This is similar in concept to the point sample rendering of Grossman and Dally [GD98]. However we use a different GPU supported depth buffer technique and propose a new splat-based merging strategy to combine framebuffer levels.

On the GPU we use a single off-screen framebuffer that is large enough to contain the images of all pyramid levels. To render into a specific pyramid level only the viewport needs to be adjusted accordingly. This way each pyramid level has its own, separate depth buffer and therefore may contain parts that will not be visible in the final image. Instead of color values, we store the points' positions and normals in the framebuffer. Additionally a radius is stored for each point. The radius r_{frag} is derived from the resolution with which the respective patch has been rendered, i.e. $r_{frag} = 2^{\max(l_{LOD},0)} r_{patch}$. This way each pixel encodes the parameters of a circular splat.

To merge the pyramid levels into a single image, the framebuffers are processed from coarse to fine. To this end the coarse level and the next finer level are bound as textures and rendered into a new texture with the same dimensions as the finer level to yield the combined image. A screen aligned quad is used to start a fragment shader for each pixel of the combined image.

The fragment shader intersects the splats encoded in the pixels of a small neighborhood in the coarse image with the ray corresponding to the combined image's pixel. The location of the intersection is used to evaluate an object-space kernel for each splat which determines its influence for the pixel at hand. We use a simple Gaussian g_σ kernel with $\sigma = \frac{1}{2} r_{frag}$. The blended contribution b from the coarse level can thus be obtained as

$$b = \frac{1}{\sum_i g_\sigma(q_i - p_i)} \sum_i g_\sigma(q_i - p_i) a_i,$$

where p_i is the position of the splat and q_i is the splat-ray intersection and a denotes any of the attributes position, normal or radius.

Since it is not guaranteed that splats from the coarse image always occlude those in the finer image, we perform a depth test before writing the blended coarse image to the result image. Should the depth test fail, the splat from the fine image is written.

The blending of pyramid levels proceeds until the finest resolution has been reached. Then, in a last step, deferred shading is applied to generate the final on-screen rendering.

7.3. Normal estimation

If the point-cloud was compressed without normal information, normals have to be estimated on-the-fly during rendering. To this end, we first render the points in the framebuffer pyramid as described above and then compute the normals in a second pass [KK05], similarly to deferred shading. This has to be done before the pyramid levels are merged in order to obtain valid splats.

In the normal estimation pass, the point positions in $5x5$ blocks of pixels are used to estimate normals. We use weighted least-squares to fit a plane to the points via PCA of the covariance matrix [HDD*92]. Again we use the object space Gaussian kernel g_σ to determine the influence of the neighbor points, where σ is computed using r_{frag} of the center pixel. Since the weights are obtained in object space, no notable smoothing occurs across edges in the image.

The normal estimation is executed once for all pyramid levels by drawing a screen-aligned quad over the whole off-screen framebuffer. After that the merging of levels proceeds just as described above.

A problem may occur during the normal estimation if areas of the object are viewed in a grazing angle. Then it can happen that neighboring pixels contain only points that are

© The Eurographics Association 2007.

model	N	R_{max}	T_D	T_{VQ}	bpp
St. Matthew	186,810,938	0.1mm	1:28	13:14	1.35 (3.31)
Atlas	255,035,497	0.1mm	2:13	15:34	1.03 (2.97)
David	28,184,526	0.1mm	0:11	1:35	1.93 (3.91)
Ephesos	23,073,902	1mm	0:16	0:05	2.37
Industrial	23,207,348	5mm	0:21	0:02	1.75

Table 1: *Compression statistics for various models. T_D gives the time for decomposition in minutes and hours. T_{VQ} is the time for vector quantization. All timings were obtained on an Intel Core 2 Duo with 2GB Ram. Bpp are measured with respect to original points. Bitrates in parentheses are for points and normals. For each model six levels-of-detail were used. For Ephesos and Industrial no normals were compressed due to their low quality.*

so distant to the center pixel's point that their weight becomes zero due to numerical reasons. In such a case the normal estimation can produce arbitrary results. While this is a principle drawback of image based normal estimation, in our case we can greatly alleviate the problem by incorporating the additional information available in the form of point normals generated from the underlying primitive shape. This normal can be used to appraise the angle under which the point is viewed and the point's radius can be enlarged accordingly. Thus, we set

$$r_{frag} = \frac{1}{< n, v >} 2^{\max(l_{LOD}, 0)} r_{patch},$$

where n is the shape normal of the point and v is the viewing direction. Since σ is directly correlated with r_{frag} the kernel width is adjusted implicitly as well. Note that the enlarged r_{frag} also increases the size of the splats used during merging of pyramid levels which fills spurious gaps between splats that can occur for steep viewing angles.

8. Results

In order to evaluate our system we conducted several experiments. Table 1 lists the bitrates achieved by our method for various models. The bitrates of our method are of the same order as results reported on smaller models by previous sequential coders. Where applicable we compressed normals with $R_{max} = 1^\circ$. In all cases we used an image tile side length of $x = 4$. Note that our method performs better on larger models as the codebook costs are better amortized. For all models, the ε parameter of the shape detection was set to equal three times the desired RMS. The parameter α was set to 30 degrees, so that parameterizable patches were found in all cases. Shape parameters and bounding boxes require between 0.2-0.3bpp and occupancy bitmaps about 0.4bpp. Only for the extremely irregularly sampled long-range scans of Ephesos and the industrial compound (see accompanying video and Fig. 7), the bitmaps take up roughly 1bpp due to the many holes and complex boundaries in the data. Due to several scanning artifacts in these scans the normals computed in a preprocess are of low quality so that they do not provide significant improvements over our screen-space estimation scheme. Thus we chose not to compress them. Also note that for this data it is extremely valuable that our method is able to adjust the sampling density locally for each patch, which is impossible to achieve with a global grid as employed by [KSW05].

In Fig. 5 a comparison of our method with the approach of Kalaiah et al. [KV05] is given. The error was measured as described in their work and the peak signal is given by the bounding box diagonal. While our method performs slightly worse for low bitrates below 0.7bpp, a high PSNR above 75 is achieved with far fewer bits. For a PSNR of 78 our system requires less than 50% bits than their method.

In order to asses the effect of the extended set of primitive shapes, we compared results of our system with all primitive shape types activated to results for which only planes were allowed. Obviously the benefit of the extended set of primitives depends on the type of geometry. On the one hand, for objects in which planar areas dominate or in which neither planes nor other shapes are actually present, none, or only a very small, gain can be achieved with the extended set of primitives (e.g. for the Michelangelo statues). On the other hand, for objects such as the oil pump (see Fig. 2) or the industrial compounds in Fig. 4 and 7 the extended set of shapes is a distinct advantage, improving the bitrate about 12%. Since planes are included in the extended set as well, we never observed an increased bitrate when all shapes were activated.

Fig. 6 shows two close-up images generated with our framebuffer pyramid. Holes are smoothly filled while detail is retained. In the image on the left normals were estimated in image-space before upsampling of the framebuffer levels. At such a scale small artifacts in the estimated normals may become visible. Fig. 8 gives some images to illustrate the performance of the normal estimation at other distances from the viewer. It can be seen that the estimated normals generally introduce a certain amount of smoothing. For larger distances this smoothing is almost equivalent to screen-space anti-aliasing but for closer views some of the detail may get lost due to the limited screen resolution. Note that detail becomes visible when the screen resolution roughly matches, or is finer than, the model resolution (see Fig. 6). The level-of-detail rendering ensures that this is mostly the case. Estimating normals in screen-space takes about 6ms per frame. In certain scenes where primitive shapes are predominant, e.g. the industrial compound shown in Fig. 7, rendering of shape normals alone already suffices to create a realistic impression.

On a GeForce 8800 GTX we currently achieve framerates between 5 to 10 fps for large models, such as Atlas or St. Matthew. We do not apply any culling techniques besides frustum culling. For some models back-face culling would result in considerable speed-up, but since many point-clouds are non-manifold (e.g. Fig. 4 and 7) back-face culling is not

© The Eurographics Association 2007.

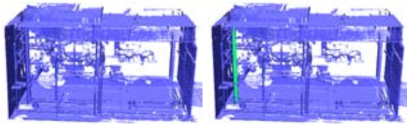

Figure 4: *Some simple interaction trivially supported in our system. A pipe is highlighted by clicking on it.*

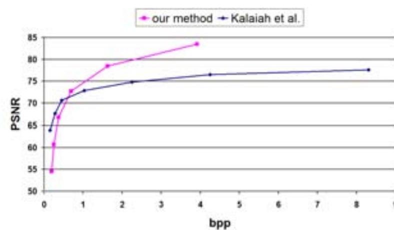

Figure 5: *The PSNR of our method compared to that of Kalaiah et al. [KV05] for the David statue.*

appropriate in general. We do plan on integrating occlusion-culling in the future. Decompression speed varies between 5-10 million points per second, depending on the levels that are decoded. Coarser levels are slower because of the render call overhead. Compared to our parallelized CPU implementation this is a speedup of about a factor of 10 (measured on an Intel Core 2 Duo).

Fig. 4 shows a small example of the interaction that is supported by our compression format. Parts corresponding to shapes can easily be suppressed or highlighted for visualization purposes. Moving or duplicating such parts would be possible as well, but we have not implemented this form of interaction yet.

9. Conclusion

We have presented a progressive compression scheme for point-clouds that aims for fast parallel decompression while achieving lower bitrates than other state-of-the-art compression algorithms which aim at direct rendering. We have demonstrated that the decompression can be executed well on today's GPUs, enabling inspection of the compressed geometry in interactive rendering. In order to support efficient parallel processing several compromises in the layout of the compressed format were made. For instance in the quad-tree hierarchy every node redundantly stores a pointer to its parent so that nodes can be processed independently. We intentionally do not make use of arithmetic or Huffman coding since streams generated with these techniques cannot straightforwardly be decompressed in parallel. Consequently, our method could achieve even lower bitrates if parallelism were not required. But even as is, our current system's compression rate on large models is of the same order than that of previous sequential coders.

9.1. Limitations and future work

The level-of-detail obtained with our approach is not suitable for very far objects, i.e. objects filling only a couple of pixels on the screen. This is due to the fact that no filtering takes place across shape borders and the number of patches is not reduced for distant views. Our approach could be extended by adding volumetric hierarchies for very coarse views, resulting in a structure reminiscent of e.g. VS-Trees [BHGS06]. Exploring these possibilities is a main avenue of future work. Also, despite the considerable accelerations proposed in this work, the runtime of the vector quantization still poses a problem for very large point-clouds. Future work will have to further address this issue. We also plan to accelerate the rendering by incorporating occlusion culling and reducing the render call overhead for coarse levels by combining different patches in a hierarchy.

Acknowledgements

The oil-pump model in Fig. 2 appears courtesy of INRIA and ISTI by the AIM@SHAPE Shape Repository. The St. Matthew, Atlas and David models are a courtesy of the Digital Michelangelo Project, Stanford University. The Ephesos data was kindly provided by Michael Wimmer.

References

[AA03] ADAMSON A., ALEXA M.: Ray tracing point set surfaces. In *SMI '03: Proceedings of the Shape Modeling International 2003* (Washington, DC, USA, 2003), IEEE Computer Society, p. 272.

[AB81] ADELSON E. H., BURT P. J.: Image data compression with the laplacian pyramid. In *Pattern Recognition and Image Processing* (1981), pp. 218–223.

[BAC96] BEERS A. C., AGRAWALA M., CHADDHA N.: Rendering from compressed textures. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), ACM Press, pp. 373–378.

[BHGS06] BOUBEKEUR T., HEIDRICH W., GRANIER X., SCHLICK C.: Volume-surface trees. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2006) 25*, 3 (2006), 399–406.

[BK03] BOTSCH M., KOBBELT L.: High-quality point-based rendering on modern gpus. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications* (Washington, DC, USA, 2003), IEEE Computer Society, p. 335.

[BWK02] BOTSCH M., WIRATANAYA A., KOBBELT L.: Efficient high quality rendering of point sampled geometry. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering* (Aire-la-Ville, Switzerland, Switzerland, 2002), Eurographics Association, pp. 53–64.

© The Eurographics Association 2007.

[CDF92] COHEN A., DAUBECHIES I., FEAUVEAU J. C.: Biorthogonal bases for compactly supported wavelets. *Comm. Pure & Applied Math 45* (1992), 485–560.

[GD98] GROSSMAN J. P., DALLY W. J.: Point sample rendering. In *Rendering Techniques* (1998), Drettakis G., Max N. L., (Eds.), Springer, pp. 181–192.

[GG92] GERSHO A., GRAY R. M.: *Vector Quantization and Signal Compression*. Kluwer Academic, Boston, 1992.

[HDD*92] HOPPE H., DEROSE T., DUCHAMP T., MCDONALD J., STUETZLE W.: Surface reconstruction from unorganized points. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1992), ACM Press, pp. 71–78.

[HH88] HANG H.-M., HASKELL B.: Interpolative vector quantization of color images. *IEEE Transactions on Communications 36* (April 1988), 465–470.

[HPKG06] HUANG Y., PENG J., KUO C.-C. J., GOPI M.: Octree-based progressive geometry coding of point clouds. In *Symposium on Point-Based Graphics 2006* (July 2006), Botsch M., Chen B., (Eds.), Eurographics.

[IK01] IVANOV D., KUZMIN Y.: Spatial patches - a primitive for 3d model representation. *Computer Graphics Forum 20* (September 2001), 511–521(11).

[KK05] KAWATA H., KANAI T.: Direct point rendering on GPU. In *Advances in Visual Computing* (2005), pp. 587–594.

[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).

[KV05] KALAIAH A., VARSHNEY A.: Statistical geometry representation for efficient transmission and rendering. *ACM Trans. Graph. 24*, 2 (2005), 348–373.

[LBG80] LINDE Y., BUZO A., GRAY R. M.: An algorithm for vector quantizer design. *IEEE Trans. on Communications COM-28*, 1 (Jan. 1980), 84–95.

[LGK*01] LENSCH H. P. A., GOESELE M., KAUTZ J., HEIDRICH W., SEIDEL H.-P.: Image-based reconstruction of spatially varying materials. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques* (London, UK, 2001), Springer-Verlag, pp. 103–114.

[LH96] LEVOY M., HANRAHAN P.: Light field rendering. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), ACM Press, pp. 31–42.

[MSM04] MATTSON T., SANDERS B., MASSINGILL B.: *Patterns for Parallel Programming*. Addison-Wesley Longman, Amsterdam, Sept. 2004.

[OH06] OCHOTTA T., HILLER S.: Hardware rendering of 3d geometry with elevation maps. In *SMI '06: Proceedings of the IEEE International Conference on Shape Modeling and Applications 2006 (SMI'06)* (Washington, DC, USA, 2006), IEEE Computer Society, p. 10.

[OS04] OCHOTTA T., SAUPE D.: Compression of point-based 3d models by shape-adaptive wavelet coding of multi-height fields. In *Proceedings of the Eurographics Symposium on Point-Based Graphics* (June 2004), pp. 103–112.

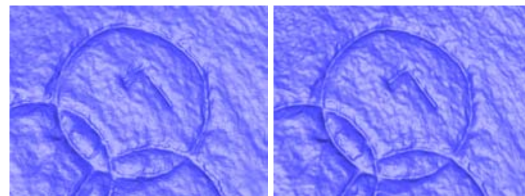

Figure 6: *Close-up of fine detail on Michelangelo's Atlas. Hole free rendering is achieved with our framebuffer pyramid. On the left decompressed normals are used. On the right normals have been estimated in screen space.*

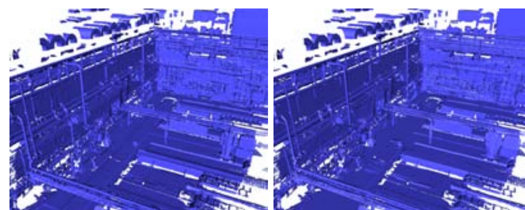

Figure 7: *The image on the left has been rendered with normals estimated in screen-space. On the right only shape normals are shown.*

[PG01] PAULY M., GROSS M.: Spectral processing of point-sampled geometry. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM Press, pp. 379–386.

[RL00] RUSINKIEWICZ S., LEVOY M.: Qsplat: a multiresolution point rendering system for large meshes. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2000), ACM Press/Addison-Wesley Publishing Co., pp. 343–352.

[SK06] SCHNABEL R., KLEIN R.: Octree-based point-cloud compression. In *Symposium on Point-Based Graphics 2006* (July 2006), Botsch M., Chen B., (Eds.), Eurographics.

[SW03] SCHNEIDER J., WESTERMANN R.: Compression domain volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (Washington, DC, USA, 2003), IEEE Computer Society, p. 39.

[SWK07] SCHNABEL R., WAHL R., KLEIN R.: Efficient ransac for point-cloud shape detection. *Computer Graphics Forum 26*, 2 (2007), 214–226.

[WGE*04] WASCHBÜSCH M., GROSS M., EBERHARD F., LAMBORAY E., WÜRMLIN S.: Progressive compression of point-sampled models. In *Proceedings of the Eurographics Symposium on Point-Based Graphics* (June 2004), pp. 95–102.

[ZPvBG01] ZWICKER M., PFISTER H., VAN BAAR J., GROSS M.: Surface splatting. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM Press, pp. 371–378.

© The Eurographics Association 2007.