GPU-Friendly High-Quality Terrain Rendering

Jens Schneider
Computer Graphics and Visualization Group
Technische Universität München
Lehrstuhl I15, Boltzmannstraße 3
D-85748 Garching bei München
jens.schneider@in.tum.de

Rüdiger Westermann
Computer Graphics and Visualization Group
Technische Universität München
Lehrstuhl I15, Boltzmannstraße 3
D-85748 Garching bei München
westermann@in.tum.de

ABSTRACT

We present a LOD rendering technique for large, textured terrain, which is well-suited for recent GPUs. In a preprocess, we tile the domain, and we compute for each tile a discrete set of LODs using a nested mesh hierarchy. This hierarchy can be encoded progressively. At run time, continuous LODs can simply be generated by interpolation of per-vertex height values on the GPU. Any mesh re-triangulation at run-time is avoided. Because the number of triangles in the mesh hierarchy is substantially decimated and by progressive transmission of vertices, our approach significantly reduces bandwidth requirements. During a typical fly-over we can guarantee extremely small pixel errors at very high frame rates.

Keywords

Terrain rendering, hierarchical meshing, GPUs, progressive data transfer, geomorphs

1. INTRODUCTION

Despite the advances in CPU and GPU technology, for the largest available terrain data sets rendering techniques still cannot run at acceptable rates *and* quality. As processing, memory, and bandwidth capabilities continue to increase, so does the resolution of scanned landscapes and recent display technology. Today, satellite range scans comprised of over a billion of samples are available, making even the handling of such data sets difficult to perform due to memory constraints. In addition, high resolution displays of about 10 Mpixels [IBM] demand a substantial increase in the number of primitives to be transferred to and processed on the GPU. The requirements imposed by current and future data acquisition and display technology make real-time visualizations difficult to perform on even the most powerful workstations. Therefore, the need for a terrain rendering system that comprehensively addresses the aforementioned issues is clear.

Journal of WSCG ISSN 1213-6972, Vol.14, 2006
Plzen, Czech Republic.

2. RELATED WORK

From a high-level view, previous approaches for terrain rendering can be classified into the three following categories.

View-dependent refinement

View-dependent refinement methods construct a continuous LOD triangulation on the CPU with respect to a given world- and screen-space error. Gross et al. [GGS95] employ a wavelet decomposition to generate adaptive quadtree meshes for terrain data, combined with a lookup-table to store an irregular triangulation for each of the possible quadtree leafs. Pajarola [Paj98] introduced *restricted* quadtrees [HB87] for terrain rendering. Duchaineau et al. [DWS+97] used triangle bintrees to perform the remeshing. Triangulated irregular networks (TINs) where first proposed by Peucker et al. [PFL78], and later automated by Fowler et al. [FL79]. Garland et al. [GH95] employed a greedy insertion strategy to construct a TIN. Progressive meshes (PMs) were modified with respect to the demands in terrain rendering by Hoppe [Hop98].

To speed up the remeshing process, frame-to-frame coherence can be exploited. Priority queues that can be updated incrementally to guide the remeshing are one alternative [DWS+97]. A different approach updates a quadtree data structure incrementally to keep track of vertex dependencies [LKR+96]. Hoppe proposed a method that keeps active cuts to achieve an incremental update [Hop98]. While the exploitation of frame-to-frame coherence usually results in a reason-

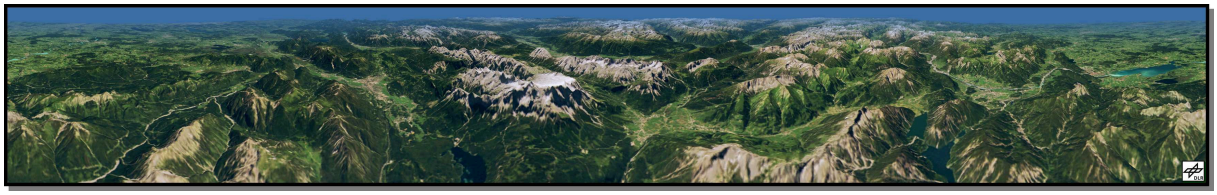

Figure 1: A 360° panorama of the Alps (7K×1K pixels), generated with our method in less than 4 seconds. This time includes rendering, reading data back from the GPU, and writing the final image to the disk.

able speed up, for particular camera movements such as shoulder views in an airplane simulation a considerable loss in performance can be observed. Furthermore, frame-to-frame coherent approaches are usually harder to implement due to LOD constraints. This was recognized by Lindstrom et al. [LP01, LP02], who proposed a simple to implement, yet efficient method to rebuild the mesh from scratch in every frame. They improve the error metric proposed by Blow [Blo00].

If the terrain gets excessively large, many of the mentioned algorithms choose to partition the terrain into square blocks or chunks of data, which can then be processed independently from each other [KLR+95, SN95]. The advantage is that these chunks can also be paged independently. However, care has to be taken to avoid invalid vertices (so-called T-vertices) at chunk boundaries. One elegant approach to avoid these invalid vertices in a quadtree was taken by Röttger et al. [RHSS98]. By restricting the error metric, they automatically guaranteed a valid mesh. However, a generalization to chunked meshes is not trivial and would also limit the error metric to a Manhattan distance.

More recently, Ulrich [Ulr00] suggested to use restricted quadtree meshes without boundary constraints for the chunks, and to fill possible cracks between them using *flanges* or *skirts* – fins of geometry along the boundaries pointing downwards from the terrain. However, ensuring correct anisotropic texture filtering at these boundaries is not trivial due to the different viewing angle. A more general approach is to stitch boundaries together using so-called zero-area triangles (also called *ribbons* in [Ulr00]), which guarantees correct filtering.

Pomeranz [Pom00] suggested to use clusters of ROAM triangulations (RUSTiC). To ensure validity, clusters are enforced to uphold an edge constraint: on shared edges the clusters must share vertices exactly. This approach is also one of the first terrain rendering algorithms exploiting graphics hardware. RUSTiC achieves improved performance over ROAM by rendering clusters as triangle strips. Hwa et al. [HDJ04] used 4-8 meshes that induce a diamond-based hierarchy on both textures and the height field. Combined with a space-filling curve memory layout this allows for efficient out-of-core rendering of the terrain, utilizing GPU memory as a cache. However, since each other texture level is rotated by 45°, a costly update of vertex texture coordinates has to be performed.

Pre-computed geometry batches

Based on the observation that on recent GPUs the time that is saved by rendering less triangles due to adaptive re-triangulation is entirely amortized by the time needed to perform the re-triangulation, several authors suggested to pre-triangulate the input data as much as possible. Cignoni et al. [CGG+03a] suggested to replace triangles in the remeshing process by a *batch*, a new primitive that approximates the terrain over a triangular part of the input domain using a pre-computed TIN. Stripping these TINs prior to rendering made them highly efficient. Batches were kept in a bintree, for which usual run-time re-meshing is performed, hence the name of the method: Batched Dynamic Adaptive Meshes (BDAM).

In [CGG+03b], the authors improved on their previous work to successfully render planet-size meshes at interactive rates. Their system does not support geomorphs, but a screen-space error of one pixel for a 640×480 view port can usually be guaranteed. However, this could become a problem soon, as displays are about to reach 10Mpixels. Consequently, considerably more triangles would have to be rendered to meet a given screen-space error.

Non-adaptive triangulation

Only very recently, Losasso et al. [LH04] took full advantage of the speed of current consumer class GPUs. They abandoned any view-dependent remeshing in favor of so-called geometry clipmaps, a triangulation that is approximately screen-space uniform. Specifically they used concentric, uniformly tessellated, square patches around the camera dropping exponentially in resolution with distance. During run time, geometry is fetched from a toroidal buffer residing on the GPU. The update of this buffer is done by the CPU.

Since the heighfield raster data is used directly, it can be compressed very efficiently. By applying a compression scheme derived from Microsoft's WMV format [Mal00], compression ratios of up to 100:1 can be achieved. Because decoding the compressed data puts a considerable amount of work on the CPU, the

decoder can eventually fall behind faster camera motions, resulting in a blurry representation of the terrain. Despite the fact that geomorphs are not an issue for this system, both the screen-space and world-space errors are hard to control, implying an rms of about 1.5m. Optimal geometry filtering cannot be performed due to the screen-space aligned topology. Also, since height fields compress a lot better than regular images, the application of photo textures will most likely result in a major increase in memory requirements. Still, extremely high frame rates for virtually arbitrarily large data sets can be achieved using this method.

3. CONTRIBUTIONS

In this work, we combine the advantages of continuous LOD semi-regular meshes with the advantages of a discrete LOD hierarchy, thus avoiding any re-triangulation at run-time. In contrast to BDAM we also avoid expensive irregular triangulations, greatly improving pre-processing from several hours to some minutes. The proposed method generates high quality renderings by supporting a continuous LOD representation including geomorphs and photo-texturing. In contrast to previous methods, the terrain is guaranteed to be refined within a user-defined screen- *and* world-space error. Aliasing is avoided by employing optimal geometry filtering, at the best possible geometric resolution. At run-time, discrete sets of decimated mesh structures are transmitted progressively, resulting in high bandwidth efficiency. To obtain a continuous LOD representations, these sets are interpolated and rendered using functionality on recent GPUs.

Algorithm overview

The domain is first partitioned into a set of equally sized tiles. For each tile, a discrete set of LODs is computed by means of a nested mesh hierarchy. The construction of such a hierarchy is described in section 4. This hierarchy has several beneficial properties: Firstly, for each level of the mesh the terrain is decimated according to a given world-space error, reducing the total amount of triangles. Secondly, to compute a continuous LOD representation, vertices at finer resolutions only have to be morphed in height onto the next coarser level. Third, as the hierarchy is nested, each finer level is represented by all vertices at coarser levels plus additional vertices required to resolve the current level properly. These additional vertices can be transmitted progressively.

The terrain hierarchy, including per-vertex morph values, is then prepared for rendering on the GPU. The particular data structure used is discussed in-depth in section 5. For textures, the S3TC standard is employed, which enables high-resolution mipmaps to be used. All data is stored in vertex buffers and 2D textures that are handled by a memory manager to minimize bus transfer. This issue is subject of section 6.

4. NESTED MESH HIERARCHY

The most common way to avoid sampling artifacts in terrain rendering is by means of a LOD representation. Such a hierarchy can either be represented implicitly by adaptive re-triangulation at run time, or it can be explicitly pre-computed for discrete LOD levels.

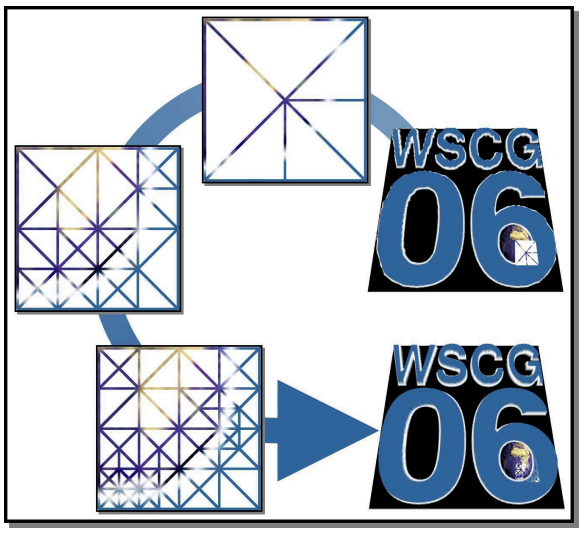

Figure 2: Levels of the nested mesh hierarchy.

A given height field $H : \mathbb{N}^2 \mapsto \mathbb{Z}$ can be approximated by a triangular mesh parameterized over a 2D domain. The surface of this mesh defines a reconstruction H' of H. The approximation quality of the mesh is then measured by a point-wise error metric $\delta : \mathbb{R}\times\mathbb{R} \mapsto \mathbb{R}$, extended to the entire domain. In the current work, we use the canonical extension of the $L_{\max}$ error metric to measure the error between H and H':

$$\delta(H,H') := \max_{x,y} \ \ \delta(H(x,y),H'(x,y))$$

By generating approximations of the height field with decreasingly lower approximation error, a mesh hierarchy that represents the original terrain at ever finer scales is constructed. The hierarchy employed in this work is *nested* with respect to the triangulation: For each triangle on level i with canonic parameterization Ω_i there is a triangle on the next coarser level $i-1$ with parameterization Ω_{i-1} such that $\Omega_i \subseteq \Omega_{i-1}$. That is, if both triangles are projected onto the domain, the triangle at level i is contained entirely in the triangle at level $i+1$. Such a hierarchy is automatically generated by restricted quadtree [HB87, Paj98], bintree [DWS+97] or red-green refinement [BSW83].

To generate a discrete set of nested hierarchy levels, the terrain is partitioned into equal tiles of size 257^2, with an overlap of one sample in either direction. Then, an error vector $(\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_{n-1})$ of exponentially decreasing entries $\varepsilon_i := 2^{n-1-i}$ is built, where the ε_i are

usually measured in meters or feet. The particular choice is motivated in section 5. Starting with ε_0, a hierarchy $\{\mathcal{M}_i\}_{i=0}^{n-1}$ of restricted quadtree meshes satisfying $V_i \subseteq V_{i+1}$ and $\varepsilon_{i+1} \leq \delta(H'_i, H) \leq \varepsilon_i$ is constructed. Here V_i and V_{i+1} are the sets of vertices at hierarchy levels i and $i+1$. More precisely, in a top-down approach we construct each $\mathcal{M}_{i+1}$ by refining $\mathcal{M}_i$, and we stop the construction if $\delta(H'_{i+1}, H) \leq \varepsilon_{i+1}$.

To generate the next finer hierarchy level from the current mesh, recursive quadtree refinement is performed until one of the following two conditions is met.

1. the maximum deviation between the new mesh and the original terrain is less than the error threshold defined for the level.
2. the spacing between vertices of the mesh becomes smaller than the error threshold defined for the level.

The second criterion is enforced by prohibiting the quadtree from being refined below a certain scale. This weakens the requirement $\varepsilon_{i+1} \leq \delta(H'_i, H) \leq \varepsilon_i$, but generally $\delta(H'_i, H)$ is still less than ε_i. In this way we can avoid aliasing artifacts due to subsampling along the domain axes. In a second step (following the Push/Pull paradigm), geometry changes are propagated from fine to coarse and sub-quadtrees are refined where needed to avoid T-vertices.

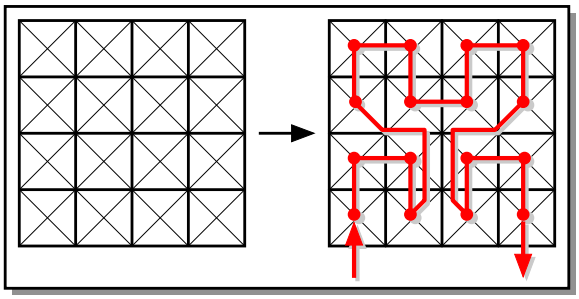

Figure 3: Quadtree mesh and Π-order traversal.

The quadtree is then decomposed into recursive triangle fans [RHSS98] or a single triangle strip [LP02]. Using triangle strips is possible in our framework, but generating them increases the time spend for pre-processing considerably. Triangle fans, on the other hand, are easy to implement, reduce meshing time and are similarly cache friendly as strips. However, generating fans results in a lot of separate primitives. In order to render these primitives efficiently, primitive restarts are employed. Primitive restarts are available on recent nVidia GPUs and are exposed in OpenGL by the GL_primitive_restart_NV extension. When rendering indexed vertices, the user may define a special index. Whenever this index is encountered, no vertex is fetched but instead a new primitive is started. This allows for a list of fans to be rendered efficiently by using only a single draw call, reducing state changes and setup overhead. To generate fans the quadtree is traversed recursively in depth-first order. As a result, we visit each fan in the order of a Π-order space-filling curve (see figure 3), which was successfully used in [LP02] to linearize memory layouts. This traversal has the nice property that fans generated after each other have a very high probability to be adjacent (in a full quadtree all consecutive fans are adjacent), in which case the newer one can re-use two or even three vertices of the previous one. Since each fan has at most 9 vertices, the last fan will always be cached entirely on current GPUs.

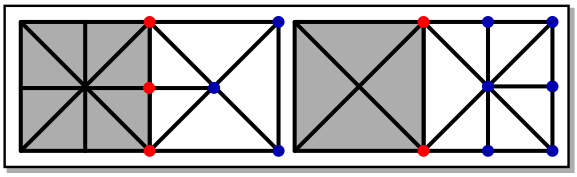

Figure 4: Best and worst cases for vertex cache re-usal of fans. The gray fan can re-use the red vertices of the white fan, resulting in a cache coherence of at least 25%

Thus, recursive fans can re-use between $2/8$ and $3/6$ of their vertices (see figure 4).

To obtain a continuous LOD representation, we interpolate between the discrete LODs $\mathcal{M}_i$. This is known as *Geomorphing* [FEKR90]. In a nested hierarchy, vertices retain their position within the domain during morphing. Due to the property $V_i \subseteq V_{i+1}$ each vertex at level i thus stores one height value for level i and each coarser level $k < i$. To render a LOD between two consecutive levels, the triangle mesh at the finer level is rendered and vertices are morphed accordingly. Although higher order interpolation is possible, only linear interpolation is considered in this work for efficiency reasons. This is described later in more detail.

5. RENDERING FRAMEWORK

As a benefit of the nested mesh hierarchy, tiles can be uploaded progressively to the GPU. On the GPU, an appropriate data structure accommodates real-time rendering at high quality, including photo-texturing. Optionally, if high resolution view ports require the screen space error to be increased, geomorphing is performed on the GPU. At the same time, the CPU performs view frustum culling and level of detail computations on a per-tile basis. Since all GPU tasks are programmed in a high-level shading language, the entire framework is extendable and can easily be tailored to fit custom needs.

GPU data structures

As soon as a particular tile has to be rendered, a vertex buffer large enough to store all shared vertices of that

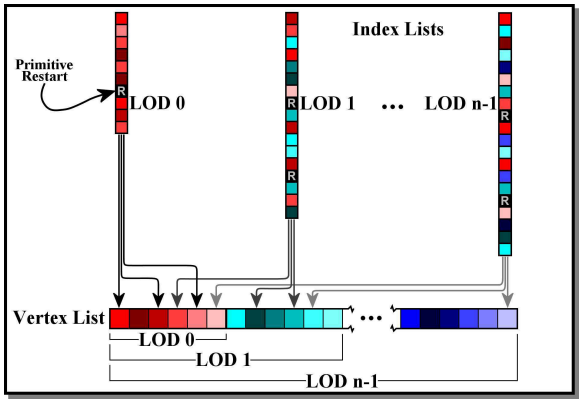

Figure 5: The GPU data structures used to enable progressive transmission of vertices and indices.

tile is created. In this buffer, vertices are organized in blocks according to their respective hierarchy levels. (see figure 5). The associated topology is stored in one separate element array for each level. The i^{th} element array contains only indices into the first $i+1$ blocks of the vertices. Such a shared vertex representation has two major advantages. Firstly, it reduces storage requirements compared to non-shared representations. This is of special importance when additional vertex attributes, such as geomorphs have to be stored. Secondly, it enables progressive transmission by re-using vertices of coarser levels.

Because the tiles used in this work always have a resolution of 257^2, relative domain coordinates are encoded in 9 bits. The height value can be considerably larger. It is therefore encoded using 14 bits. All three values are stored in two 16 bit vertex attribute components. They are decoded in the vertex shader during rendering.

If geomorphs are enabled, additional storage requirements arise. The method is still memory efficient, as only one additional height value per coarser level needs to be stored. Since usually only small offsets to the original height are needed, 8 bits per value are sufficient. This allows us to morph vertices within a range of $+127\ldots-128$ units.

Run time processing

For each tile we keep an axis-aligned bounding box to accommodate view frustum culling on the CPU. For every frame, visible tiles are depth-sorted to exploit the early-depth test, if available, and to reduce overdraw. A memory manager, which is described below, ensures that all visible tiles can be rendered by paging in data not yet resident on the GPU.

Then for each visible tile the appropriate LOD is computed. The index buffer as well as the vertices required to render the respective level are sent to the GPU, if not already resident. If a tile has been rendered previously, at least a subset of vertices has already been sent to the GPU. In this case, only the remaining vertices required to render the current level are transmitted and written to the respective vertex buffer on the GPU. In this way, even though an array large enough to keep all vertices has to be allocated on the GPU, bandwidth requirements at run time are substantially reduced.

To avoid cracks at tile boundaries, neighboring tiles are stitched together using zero-area triangles. For each tile and each level in the hierarchy, the set of border vertices along with all attributes is duplicated in system memory. Whenever two neighboring tiles are visible, the necessary triangles to fill out T-junctions are generated on the CPU and are then rendered. Since this process uses exact duplicates of the vertices on the GPU and the same GPU programs are employed, cracks are avoided without numerical precision issues.

Level of detail

Determining the appropriate LOD for each tile and vertex requires the projection of the user-defined pixel error to object space. Previous approaches rely on conservative estimates of this error and are often equivalent to a linear approximation of the projection. Since such estimates usually over-estimate the error, even for pixel errors larger than one aliasing might still occur. We compute a more precise error metric by directly using the current projection matrix, which maps homogeneous object coordinates $v=(v_1,v_2,v_3,1)$ to screen-space coordinates $s=(s_1,s_2,s_3)$. Here, s_3 corresponds to the depth value. The appropriate scale of details ρ can then be computed in a similar way as the appropriate mipmap scale for texturing [Wil83]:

$$\rho := \sqrt{\frac{\sum_{i=1}^{3}\left(\frac{\partial v_i}{\partial s_1}ds_1+\frac{\partial v_i}{\partial s_2}ds_2\right)^2}{ds_1^2+ds_2^2}}$$

To compute ρ, s is expressed in parametric form $s(v)$, already including perspective division and scaling of the canonic frustum to pixel coordinates. The Jacobi matrix at v consists of the partial derivatives $J_{ij}(v) := \partial s_i/\partial v_j$. The inverse transpose of $J(v)$ contains exactly the partial derivatives required to compute ρ. The differentials ds_i are required to map from units of the height field (*eg.*,feet or meters) to pixels. Computing ρ yields the optimum scale corresponding to a screen-space error τ equal to 1 pixel. If the user selects a different screen-space error, the frustum is scaled to pixel coordinates divided by τ instead of using the entire resolution. Then, ρ is the object space distance that projects onto τ pixels.

On the CPU, ρ_j is computed per tile for each corner j of its bounding box. Because entries of the error vector are given by $\varepsilon_i = 2^{n-1-i}$ units, the optimum LOD value is computed by $\lambda_j := \lambda_{max} - \lfloor \log_2(\rho_j) \rfloor$, where

$\lambda_{max} = n-1$ is the number of available levels. The mesh $\mathcal{M}_{\min_j\{\lambda_j\}}$ is then selected for rendering the tile.

Geomorphing

As mentioned before, high resolution displays coupled with a low screen-space error can require most of the terrain to be rendered at the highest resolution. In order to maintain stable and interactive frame rates, the tolerable screen-space error has to be increased. To prevent popping artifacts, geomorphs are applied. For every vertex v in a tile, the λ_j at box corners are trilinearly interpolated on the GPU to get an approximate vertex LOD $\lambda(v)$. Geomorphing [FEKR90] now consists of linearly interpolating height values $H_{\lfloor\lambda(v)\rfloor}$ and $H_{\lfloor\lambda(v)\rfloor+1}$,using the fractional part $\lambda(v) - \lfloor\lambda(v)\rfloor$ as interpolation weight.

Finding the correct height values on the GPU could be implemented in a straight forward manner using conditionals. As conditionals are costly on current GPUs, we avoid them by implementing a different approach based on clamped forward differences. In this approach, we treat height values $\{H_i\}_{i=0}^{n-1}$ as the control points of a piecewise linear interpolant in λ. To

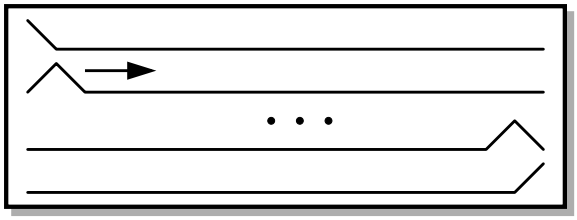

Figure 6: Basis-functions η' for geomorphs.

obtain $H(\lambda)$, we compute shifted basis-functions that can be reduced using simple dot product arithmetic. Firstly, we compute a vector-valued function
$\eta(\lambda) := clamp\left((\lambda,\lambda,\lambda,\lambda,\ldots)^t - (0,1,2,3,\ldots)^t, 0, 1\right)$

Each component i of η contains a linear ramp between $\lambda = i$ and $\lambda = i+1$. For $\lambda \leq i$ it is 0, and for $\lambda \geq i+1$ it is 1. Then, the desired basis function is obtained by computing forward differences on η:

$$\eta_i'(\lambda) := \begin{cases} 1 - \eta_0(\lambda) & \text{if i=0} \\ \eta_{i-1}(\lambda) - \eta_i(\lambda) & \text{else} \end{cases}$$

Finally, the η_i' contain the well-known basis functions for linear interpolation (see figure 6). Interpolation can now be written as the dot product $H(\lambda) = \sum_{i=0}^{n-1} \eta_i'(\lambda) \cdot H_i$. This method is highly efficient on the GPU and in our case ($n = 9$) outperformed the straight-forward implementation using conditionals by a factor of 2.5.

Texturing

By default, a pre-lit 2D texture is mapped onto the terrain. This can be a photo texture or, as for the Puget Sound, a synthesized 2D texture. During pre-processing, the texture is dyadically downsampled using a Lanczos filter with radius 2 to obtain a single, large mipmap. Now tiles are cut out of the mipmap to precisely match the tiles of our mesh hierarchy. To save GPU memory and bandwidth, each texture tile is then compressed using the S3 compression scheme. More specific, tiles are encoded using the DXT1 format, which yields good results for most photographic or synthetic textures at a compression rate of 6:1. We store the $16K^2$ Puget Sound texture including 9 (11) mipmap levels for the $16K^2$ ($4K^2$) geometry in about 170 MB.

If a pre-lit texture is not available, it is computed from the original terrain in a pre-process. Alternatively, normals could be stored as additional vertex attributes. However, besides the additional memory overhead that is introduced (at least two 8 bit values to cover the upper hemisphere), lighting artifacts due to non-continuous changes of normals during LOD transitions can only be resolved by storing one normal per vertex *and* level. On the other hand, a DXT1 pre-lit texture with 4 texels per vertex has the same storage requirements as a single per-vertex normal, but it avoids any lighting artifacts because texture filtering is performed *after* lighting.

6. MEMORY MANAGEMENT

After building the discrete LOD hierarchy, for high-resolution terrains including morph values and textures, the data is far too large to be stored in local video memory of recent GPUs. To avoid frequent paging of textures and vertex buffers, and to optimize progressive updates we have implemented a memory manager. At initialization time, the memory manager allocates chunks of exponentially growing sizes in GPU memory, to prevent external fragmentation. Sizes range from 32KB to a maximum size that allows the largest vertex buffer to be stored in such a chunk. Additionally a number of textures with a fixed resolution is allocated. The memory manager stores meta-data for each memory block, i.e. size, a time stamp, and the number of levels already sent to the GPU. Paging is now implemented as a mixture between a last recently used (LRU) and a tightest fit (TF) strategy.

Whenever a tile A is to be rendered, the system determines if there is already a chunk associated with A. If not, and also no appropriate chunk is available, the tile B with the earliest time stamp large enough to completely store A is determined. B is then marked as non-resident, and the chunk is overwritten with the data of A. To efficiently determine B, we keep a priority list for each available size. This allows us to weight the LRU strategy against a TF criterion. Once a chunk has been associated with A, all data required to render the current level is sent to the GPU. If there already was a chunk associated with A, the memory manager determines whether the chunk contains all necessary data. If not, the CPU sends all missing vertices and the

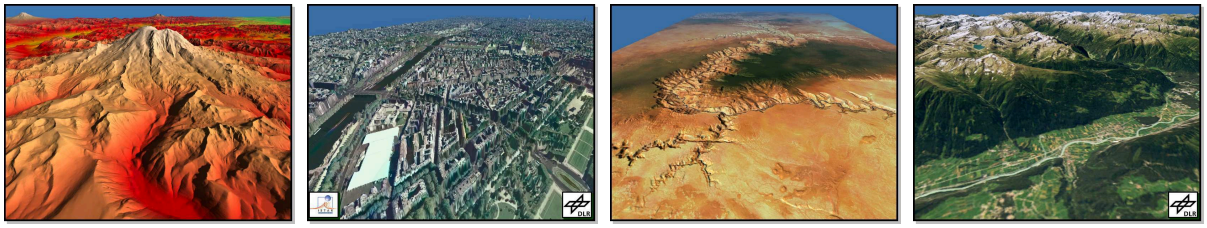

Figure 7: Test data sets in this paper. From left to right: Puget 16K×16K, Paris, Grand Canyon, and Alps. Observe the high degree of geometric details present even in regions further away from the viewer.

Data Set	Resolution	Texture	original Size	Storage	fps $\tau = 1$	MΔ/sec $\tau = 1$	fps $\tau = 5$
Puget4K	4K×4K	16K×16K	800MB	412MB	202	78.85	199
Puget16K	16K×16K	16K×16K	1.25GB	1.25GB	60	25.69	57
Grand Canyon	4K×2K	8K×4K	112MB	80MB	289	74.60	292
Paris	9.7K×5.8K	19.5K×11.7K	763MB	267MB	36	100.87	65
Alps	8.9K×8.5K	8.9K×8.5K	361MB	546MB	145	65.43	155

Table 1: Timings and Results. Original size only includes height field and texture, without taking mipmaps into account. τ refers to the pixel error. For $\tau = 1$ geomorphs were disabled, for $\tau = 5$ they were enabled.

required index buffer to the GPU. Since vertices are shared across levels, this update is usually very cheap compared to the upload of all vertices. Whenever a tile is rendered, its time stamp is updated.

The memory manager supports uniform load on the bus connecting the CPU and the GPU, thus avoiding 'paging hiccups': when a non-resident tile enters the view frustum, there is usually another one that has to be released, the texture tile has to be uploaded, and an initial LOD has to be sent to the GPU. However, with high probability this initial LOD requires only a few vertices. On the other hand, if a tile was already resident, performing an update only requires a fraction of the entire data to be sent.

Speculative prefetches are also supported, if there are unused memory chunks. If the number of chunks needed to render the current view falls below a certain fraction of all allocated chunks, the user's view is predicted. Whenever the user moves, a list containing the last viewing parameters is updated. By fitting a spline through these parameters, new viewing parameters can be extrapolated and tiles that are predicted to become visible in the near future can be prefetched, as long as a maximum time budget is not expired. In this way, very smooth fly-overs at high frame rates can be achieved.

7. RESULTS

Our results and timings are summarized in table 1. All timings were done on a P4 3.0GHz with 2GB RAM and GeForce 6800GT with 256MB. The machine was equipped with a single standard 120GB IDE hard disk. All data sets were rendered to a 1024×768 view port. Enabling 8x full-screen anti-aliasing and 4x anisotropic texture supersampling, the frame rate dropped about 30%. The timings should be fairly comparable to more recent publications. Though we have a newer graphics card, we render a considerably larger view port compared to many other systems.

Pre-processing of the geometry to a 9 level hierarchy processes approximately 15M vertices per minute and is linear in the amount of vertices. Memory consumption is constant, as tiles are processed independently of each other. Generating a 16K×16K texture hierarchy including filtering takes about 5 Minutes.

The Puget4K and the Grand Canyon data sets are only medium sized, and consequently our system is neither triangle nor memory limited. For the Paris data set with its 2.8MΔ per frame, we become triangle limited. Note however that this is a worst-case scenario, as our triangulation faithfully reconstructed all the steep sides of the buildings. A lot of these triangles are backfaces that are culled by OpenGL (but they are still counted since they pass the geometry stage). However, the Paris dataset is an excellent benchmark for the raw triangle throughput that our system can achieve.

The Puget16K dataset on the other hand is large enough to demonstrate the effects of the memory system. The lower triangle throughput reflects that our paging strategy does not come for free, but it still allows for highly interactive fly-overs

The Alps data set is a good mixture between these extremes. It contains lots of flat terrain around Munich and a considerable amount of very rough terrain around the Alps.

As our results show, frame rates for highly triangulated data sets, such as Paris, can also be improved by increasing the pixel error and enabling geomorphing. For these highly triangulated datasets we also hope to benefit from continuously increasing vertex processor throughput on future graphics chips.

8. CONCLUSION & FUTURE WORK

We have presented an efficient rendering system for large and textured terrain data that provides excellent quality and highly detailed views. In particular, at equal frame rates our system guarantees a smaller pixel error than previous approaches. We achieve these properties by exploiting a special discrete LOD hierarchy, as well as processing and rendering functionality on recent GPUs.

In the future, we will investigate dedicated compression schemes that are amenable to GPU decoding, such as vector quantization. Both, the possibility to compress mesh hierarchies as well as texture will be considered. As GPUs are become increasingly powerful, adaptive on-the-fly texture synthesis will become an important feature.

Acknowledgements

We would like to thank the DLR and ISTAR for the Paris and Alps data sets and the people from GA Tech [Geo] for making the Puget Sound and Grand Canyon datasets publicitly available.

9. REFERENCES

[Blo00] J. Blow. Terrain rendering at high levels of detail. In *Game Developer's Conference*, 2000.

[BSW83] R. E. Bank, A. H. Sherman, and A. Weiser. Refinement algorithms and data structures for regular local mesh refinement. In *Scientific Computing*, pages 3–17, 1983.

[CGG+03a] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. BDAM – batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum*, 22(3):505–514, 2003.

[CGG+03b] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Planet-sized batched dynamic adaptive meshes (p-bdam). In *Proc. IEEE Visualization '03*, pages 147–154, 2003.

[DWS+97] M. A. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *Proc. IEEE Visualization '97*, pages 81–88, 1997.

[FEKR90] R. L. Ferguson, R. Economy, W. A. Kelly, and P. P. Ramos. Continuous terrain level of detail for visual simulation. In *IMAGE V Conference '90*, pages 144–151, 1990.

[FL79] R. J. Fowler and J. J. Little. Automatic extraction of irregular network digital terrain models. In *Proc. ACM SIGGraph '79*, pages 199–207, 1979.

[Geo] Georgia Institute of Technology. Large Geometric Models Archive. http://www.cc.gatech.edu/projects/large_models.

[GGS95] M. H. Gross, R. Gatti, and O. Staadt. Fast multiresolution surface meshing. In *Proc. IEEE Visualization '95*, pages 135–142, 1995.

[GH95] M. Garland and P. Heckbert. Fast polygonal approximation of terrains and height fields. Technical Report CMU-CS-95-181, Carnegie Mellon University, 1995.

[HB87] B. Von Herzen and A. H. Barr. Accurate triangulations of deformed, intersecting surfaces. In *Proc. ACM SIGGraph '87*, pages 103–110, 1987.

[HDJ04] L. M. Hwa, M. A. Duchaineau, and K. I. Joy. Adaptive 4-8 texture hierarchies. In *In Proc. Visualization*, pages 219–226, 2004.

[Hop98] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proc. IEEE Visualization '98*, pages 35–42, 1998.

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

[KLR+95] D. Koller, P. Lindstrom, W. Ribarsky, L.F. Hodges, N. Faust, and G.A. Turner. Virtual GIS: A real-time 3D geographic information system. In *Proc. IEEE Visualization '95*, pages 94–100, 1995.

[LH04] F. Losasso and H. Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. In *Proc. ACM SIGGraph '04*, pages 769–776, 2004.

[LKR+96] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time, continuous level of detail rendering of height fields. In *Proc. ACM SIGGraph '96*, pages 109–118, 1996.

[LP01] P. Lindstrom and V. Pascucci. Visualization of large terrains made easy. In *Proc. IEEE Visualization '01*, pages 363–370, 2001.

[LP02] P. Lindstrom and V. Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):239–254, 2002.

[Mal00] H. S. Malvar. Fast progressive image coding without wavelets. In *Proc. IEEE Data Compression*, pages 243–252, 2000.

[Paj98] R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *Proc. IEEE Visualization '98*, pages 19–26, 1998.

[PFL78] T. K. Peucker, R. J. Fowler, and J. J. Little. The triangulated irregular network. In *Proc. ASP-ACSM Symposium on DTM's*, 1978.

[Pom00] A. A. Pomeranz. ROAM using surface triangle clusters (RUSTiC). Master's thesis, Center for Image Processing and Integrated Computing, University of California, Davis, 2000.

[RHSS98] S. Röttger, W. Heidrich, P. Slusallek, and H. P. Seidel. Real-time generation of continuous levels of detail for height fields. In *Proc. WSCG '98*, pages 315–322, 1998.

[SN95] M. Suter and D. Nüesch. Automated generation of visual simulation databases using remote sensing and GIS. In *IEEE Visualization '95*, pages 86–93, 1995.

[Ulr00] T. Ulrich. Rendering massive terrains using chunked level of detail. ACM SIGGraph Course "Super-size it! Scaling up to Massive Virtual Worlds", 2000.

[Wil83] L. Williams. Pyramidal parametrics. In *Proc. ACM SIGGraph '83*, pages 1–11, 1983.