RASTeR: Simple and Efficient Terrain Rendering on the GPU

Jonas Bösch[†] , Prashant Goswami[‡] and Renato Pajarola[§]

Visualization and Multimedia Lab, Department of Informatics, University of Zurich

Abstract

This paper introduces RASTeR, a GPU based LOD technique for interactive rendering of large terrains based on a paired multi-resolution tree structure. Our approach uses regular height-data blocks and terrain independent triangle patches, which are used to efficiently subdivide the terrain data. At run time, continuous LODs can simply be generated by tiling a limited set of triangle patches, the indices to which are pre-computed, over height-field blocks, thereby minimizing the amount of data to be transferred to the graphics card. RASTeR maintains a constant frame rate through asynchronous and a priori fetching of raw or compressed elevation and texture data. The efficiency of our method is validated by presenting experimental results on large elevation models.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.7]: Three-Dimensional Graphics and Realism—Computer Graphics [I.3.3]: Picture/Image Generation—

1. Introduction

Efficient real-time visualization of very large terrain data remains challenging as digital elevation models (DEMs) are reconstructed at increasingly higher resolutions. Moreover, in a number of visualization systems terrain rendering itself is only one task among many to support the display of and interaction with other data. It might even be relegated to the background. Hence further investigations are required to develop efficient algorithms that exploit the commonly available hardware as much as possible and that perform at a level which allows combination with other data in complex visualization applications. However, the most efficient algorithms to date typically do not tend to be the solutions most easy to use or implement, may require significant preprocessing and often exhibit a complex binary representation of the elevation data not amenable to any other use than rendering.

Many algorithms with different trade-offs have been proposed for interactive rendering of large DEMs, limiting the 3D graphics load effectively to the available computing and graphics resources as well as display needs. To improve rendering performance, an appropriate level-of-detail (LOD) [LRC*03] of the graphics data to be displayed is selected for each rendered frame. The LOD is adaptive with respect to surface features and viewing parameters and is selected such as to achieve a targeted rendering quality and frame rate. With current generation CPU-GPU configurations, however, LOD rendering should not be optimized based on individual geometric primitives, such as vertices or triangles, but instead should take advantage of batched graphics primitives. Failing to do so, the CPU may very easily consume too much computational time to perform such fine-grain optimization of the rendering load, risking to starve the fast-paced graphics hardware pipeline.

We have identified the following list of desirable features and properties of terrain rendering algorithms that should be supported by an effective DEM visualization system.

LOD support: Multiresolution terrain model that supports continuous adaptive LOD triangulation.
High-performance rendering: Data primitives and display units that exploit GPU accelerated rendering.
Continuous display: Asynchronous loading of data from out of core.
Fast data retrieval: Efficient caching and prefetching exploiting the multi-level CPU and GPU memory hierarchy.
Compact storage: Effective storage model and data compression for reduced memory consumption.
Direct data access: Simple and direct DEM height data access for other non-rendering usage scenarios.
Simplicity: Easy to adapt algorithms and data structures for widespread usability.
Fast preprocessing: Inexpensive data preparation and updates.

Adaptive LOD meshing and fast rendering from out-of-core is addressed by virtually all recent terrain rendering approaches. Compact storage and data compression is additionally supported by a number of methods as well. However, many proposed systems are fairly complex and often

[†] boesch@ifi.uzh.ch
[‡] goswami@ifi.uzh.ch
[§] pajarola@acm.org

feature a tightly integrated triangulation and DEM storage structure which does not support direct access to elevation data for purposes other than rendering. Therefore, the integration of an efficient terrain rendering algorithm into another system working with DEM data can require the duplication of the DEM data specifically for display purpose.

In this paper we propose a novel technique for efficient LOD-based rendering of terrain data which departs from this tight integration of LOD triangulation and terrain data storage. It features a simple tile based data format for easy elevation and texture data management. However, unlike early tile based systems, our approach as well supports continuous adaptive LOD meshing through an implicit triangulation model which depends only on a lean LOD data structure. This approach allows using various DEM compression methods, supports direct access to the DEM data through the use of standard data and compression formats, provides high-performance LOD rendering and as well features simple and fast preprocessing.

Our novel adaptive LOD rendering algorithm is a GPU intensive solution which is based on a paired multiresolution tree structure. We make use of two new concepts , *K-patches* and *M-blocks*, which are the main units of a triangle bintree and raster quadtree multiresolution hierarchy respectively. The main contributions and advantages of our LOD approach, which facilitate the desirable DEM visualization features above listed are:

1. Implicit definition of a semi-regular multiresolution triangulation based on triangle mesh patches.
2. Support for effective regular grid DEM data management independent of the multiresolution triangulation.
3. Fast adaptive LOD meshing using a batched meta bintree data structure.
4. High-performance hardware-accelerated rendering using a GPU intensive meshing solution.

2. Related Work

Level-of-detail (LOD) based polygonal meshing and multiresolution rendering has received much attention over the last decade [LRC*03]. Exploiting the regular grid structure of DEMs, multiresolution restricted quadtree or bintree approaches such as [SS92, LKR*96, DWS*97, Paj98] have shown to generally be more performant than irregular triangle mesh (TIN) based methods as proposed in [Pup96, DFMP96, Hop98]. Variations of the theme appeared as 4-8 or right-triangular meshes, and efficient out-of-core data clustering as well as effective view-dependent error metrics have been developed (see [PG07]).

Tiled block and nested regular grid approaches are simple and efficient but often not as powerful as the restricted quadtree or triangle bintree approaches we are focusing on here. For further details and comparisons of the different related techniques we refer the reader to the survey [PG07].

As current graphics hardware can render many millions of triangles per second, the CPU-GPU communication has become a major bottleneck. To avoid starvation from a slow CPU process selecting a costly fine-grain LOD mesh to be rendered by the (faster) GPU, block-based LOD selection and terrain rendering techniques have been proposed. RUSTIC [Pom00] and CABTT [Lev02] improve the ROAM approach [DWS*97] by coarse-grained on-board caching of static and dynamic triangle clusters respectively. Triangle clusters or blocks also form the basic unit of LOD refinement in [CGG*03, LPT03, HDJ05]. These methods demonstrate the performance benefits of coarse LOD adaptation despite the increased number of per-frame rendered triangles.

We can observe that with current rendering rates, it is now possible to render adaptive scenes with triangles that have a projected screen size of one or only a few pixels. At this point, it is no longer beneficial to carefully choose an optimal fine-grain set of LOD triangles, but it is more economical to perform a quick LOD selection of patches of triangles at a coarser grain. This basic idea is followed by the block-based techniques outlined above. In contrast to the previous approaches, our solution introduces an implicitly defined multiresolution triangulation of triangle mesh patches, and it separates the DEM's height data management from the LOD organization, which allows for efficient multilevel grid based out-of-core data structures.

3. RASTeR Principles

Our real-time adaptive simplification and terrain rendering (RASTeR) system presents a novel theoretical and practical framework for patch-oriented multiresolution triangulation and rendering of grid-based digital terrain elevation models (DEMs). For this, two new conceptual triangulation and data units are defined, *K-patches* and *M-blocks*.

3.1. Adaptive Triangulation

3.1.1. K-Patches

A semi-regular restricted quadtree, bintree or 4-8-mesh LOD triangulation is given by a manifold mesh of conforming isosceles triangles such that adjacent triangles differ by at most one level in the LOD hierarchy. The hierarchical LOD triangle mesh representation can be viewed as a binary tree, recursively splitting isosceles triangles at their longest edge (see also Figures 3 and 5(b)). Cracks between adjacent triangles of different LODs are avoided by always refining both triangles sharing the longest edge that is split. This may cause force-propagated splits to neighboring triangles and results in neighboring LOD triangles differing by at most one level [SS92, LKR*96, DWS*97, Paj98].

The concept of using triangle clusters for terrain rendering has been introduced in [Pom00, Lev02, CGG*03, LPT03]. Along the lines of [HDJ05] we define regularly triangulated clusters with a constant number K of vertices along each triangle-patch edge, see also Figure 1. As we will see later in Section 4, however, our K-patches are never explicitly stored, and theoretically K could be chosen differently each time at startup of the visualization application (but should be a power of two plus one).

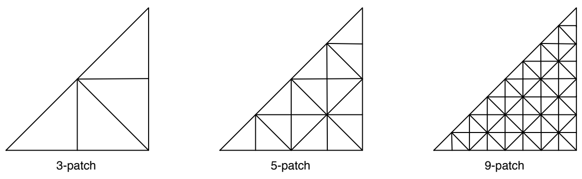


Figure 1: *Triangle K-patches for different sizes of K.*

Following the constraints on the bintree triangle subdivision, an adaptive batched LOD triangulation can be achieved by selectively splitting the K-patches (see also Figure 5(b)). Doing this recursively starting from a coarse set of (two) initial K-patches, the resulting set of variably sized K-patches can be interpreted as macro triangles of a batched *meta* bintree, and thus could be arranged in a triangle strip sequence. In fact, also the interior triangulation of a K-patch can be represented by a single triangle strip, see also [PG07].

The orientation of a K-patch is always an instance of one of eight basic isosceles triangle types, as shown in Figure 2. According to the bintree multiresolution model, a K-patch can be subdivided yielding two child K-patches whose type is dependent only on the parent, as shown in Figure 3.

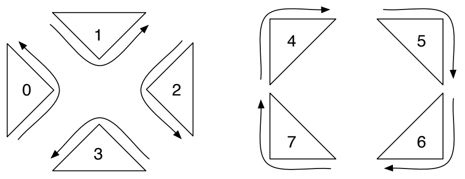


Figure 2: *The eight basic orientations of K-patch types.*

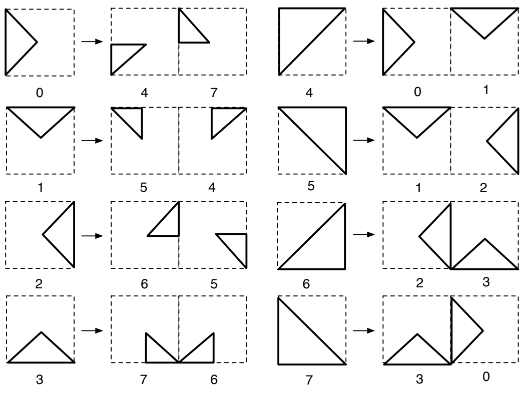


Figure 3: *K-patch parent-child orientation transition cases.*

Note that each K-patch is defined to have an associated orientation, indicated in Figure 2. This direction specifies the numbering of the K-patch's vertices. Also the two children of a K-patch split are defined to be numbered in the order of the parent direction. That is, the child starting at the first vertex is referred to as child 0, the other as child 1.

K-Patches are organized in a *meta* bintree so that each K-patch represents a node, similar as in [CGG*03] (see also Figure 5). Since K-patches of types $0,1,2,3$ share the same vertex spacing as their children, the resolution is doubled for every two consecutive levels in the meta bintree, as can also be seen in Figure 4(a). This is important to note for the relationship of K-patches and M-blocks, as discussed below. Cracks between adjacent K-patches of different LODs are avoided by the splitting constraints outlined at the beginning of this section. An elegant method to avoid forced splits is to saturate the LOD error metric [Paj98, OR99, Ger99]. On our meta bintree of triangle K-patches, the saturated view-dependent error metric is implemented as in [LP01, Ger03].

3.1.2. M-Blocks

An M-block is a square block of a regular grid of height sample data – and possibly other attributes such as surface normal – stored in a file on disk. All M-blocks are defined to be of equal size, that is, they have the same number $M \times M$ of vertices with $M = 2^m + 1$, being a power of two plus one for consistent overlap between blocks. M-blocks are organized in a quadtree hierarchy, with each M-block representing a node, also storing its scale factor and bounding box. In this reduced resolution pyramid [Wil83], the resolution of the terrain changes by a factor of two between levels.

Currently, the M-block data is tested both in uncompressed and reversibly compressed JPEG2000 format, but other compression methods could be used. Decompression of the M-block data can be done asynchronously by streaming the data into OpenGL Pixel Buffer Objects, which are a superset of the Vertex Buffer Objects (VBO) normally used to store M-blocks on the GPU.

3.1.3. M-Block – K-Patch relationship

M-blocks and K-patches form two separate but tightly connected hierarchies, and thus each meta bintree node stores a pointer to the associated quadtree node, see also Figure 5. Exploiting this relationship means that it is possible to clearly separate the LOD selection from the rendering, both conceptually as well as in the implementation. The LOD selection is performed on the K-patch bintree, considering a K-patch as a triangle of triangles, while rendering and resource management are done on the M-block quadtree, considering K-Patches as triangle strips over M-block vertex buffers.

Given M being a small multiple f of K, we note that within a single square M-block we can form $2 \cdot f$ different LOD levels of K-patch triangulations. Furthermore, the K-patch based triangulation is aligned with M-block boundaries, at varying resolutions. Hence any of the basic K-patch types can be placed at a fixed number of positions and scales within a given M-block. Figure 4(b) shows the possible scalings and alignments for a K-patch of type 1 in an M-block with $M = 2K$. Since the combinatorial possibilities of scaling and placing the basic K-patch triangle patches within an M-block of given size are limited, we can precompute these configurations and index them in a table.

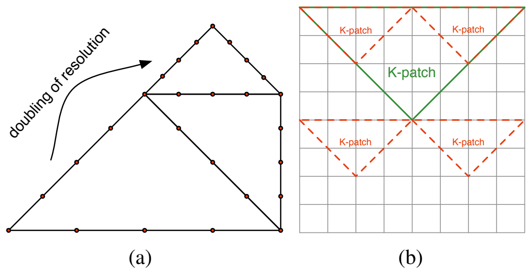


Figure 4: *(a) Every two levels of K-patch subdivision the resolution doubles. (b) Placings of a type-1 K-patch in an M-block of size* $M = 2K$*.*

3.2. Terrain Level-of-Detail

As error metric we adopt a saturated view dependent definition as in [LP01, Ger03]. It is based on a basic object space error, the difference in height of the refinement vertex to the midpoint of the longest edge of a triangle. However, instead of defining the LOD error on each vertex, which would be wasteful considering the batch-based approach taken in this algorithm, it is only defined for the K-patch refinement points. The correctness of the error is still guaranteed for K-patches of any size because of the saturation property. In fact, by changing the size of a K-patch, the granularity of the LOD selection can be adjusted, but other factors such as preferred (GPU-optimal) rendering batch size usually take precedence for choosing K.

A meta bintree is used for adaptive multiresolution triangulation. In this binary tree, each node corresponds to a triangle K-patch of a certain type and scale, and it stores the LOD error of the K-patch's refinement vertex. The node furthermore maintains a pointer to the corresponding M-block of height values and the K-patch's offset within it.

3.3. GPU Meshing

We note that for a given K-patch type, the indices of its triangle-strip vertices relative to the corresponding M-block are invariant up to a scale factor and offset of the K-patch within the M-block, see also Figure 4(b). Therefore, we initialize the set of index arrays for all K-patch types using OpenGL Element Array Buffers on the GPU to be used for indirect indexing into the M-block DEM height data. Given

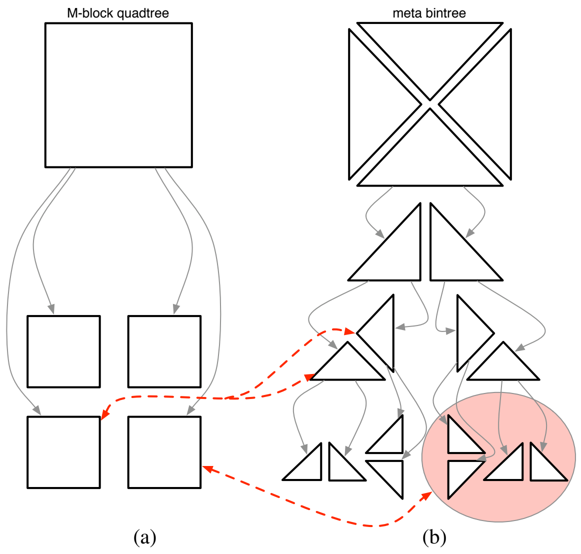


Figure 5: *M-block height-field quadtree nodes (a) can correspond to different K-patch meta bintree nodes (b) depending on the selected LOD triangulation. Elevation data in the M-block quadtree is separated from the triangle mesh connectivity in the K-patch meta bintree.*

the M-Block's base or anchor point p_0, K-patch offset off and scale factor s, the vertices p_i of the the patch's triangle strip in 2D can be generated on a shader as follows:

$$p_i = p_0 + off + (s \cdot x_i, s \cdot y_i), \qquad (1)$$

where x_i and y_i are the indices of the i-th vertex in the triangle strip of a K-patch, relative to the patch's start vertex. Correspondingly, the DEM's height value is fetched from the current M-block heigh-mapt at entry $off + (s \cdot x_i, s \cdot y_i)$.

As outlined above, each triangle K-patch in the LOD meta bintree is associated with an M-block. Hence for a given LOD selection a number of M-blocks are *active*, and each may be referenced by one or more non-overlapping triangle K-patches. The active M-blocks are dynamically cached as VBOs on the GPU and bound at run-time. Rendering a K-patch triangle strip corresponds to identifying the K-patch type and sending its index, scale and base offset to the GPU.

4. RASTeR Construction

4.1. DEM Preprocessing

In the preprocessing step, the source DEM data has to be parsed and converted into a simple intermediate representation. This representation consists of standard grid-digital terrain height-field data plus additional multiresolution LOD information. The preprocessing steps are quite simple and can be carried out in a short time.

For each M-block, the basic elevation data is maintained in a simple grid-digital height-field format. Besides the altitude value for each height sample, another attribute that can (optionally) be stored is the surface normal. The data of each M-block can easily be stored in a separate file as it corresponds to the basic unit of DEM data access from disk. In fact, to further optimize I/O performance, multiple M-blocks could be aggregated into larger blocks of height-field data with generation of multiple resolutions of M-blocks on the fly from the larger aggregated blocks on disk.

This simple DEM data management in form of height-field grids allows other applications or functions to directly access the elevation data for other purposes than display without having to extract it from a more complex and integrated hierarchical multiresolution triangle mesh representation. Since M-blocks or larger aggregated height-field data blocks are not restricted to a particular grid-digital data format, various source formats can be supported. Hence to save disk storage M-blocks could be loaded or extracted from compressed height-field data. JPEG2000 and other simple compression methods have been implemented in the RASTeR system as a proof of concept. However, to avoid discontinuities at block boundaries from different lossy compression in adjacent blocks, boundary consistency must be enforced through modified compression, explicit storage of the boundary samples, or on-the-fly adjustments at runtime. In general, a simple M-block access layer can support various underlying binary and compressed height-field data formats.

The complete K-patch information consists of data-independent triangle-strip index configurations, as well as of the meta bintree with per-node LOD error metric and M-block attributes. The geometric object-space error is computed for each height-field vertex and saturated bottom-up. Each meta bintree node, corresponding to a particular K-patch triangle type, stores the saturated LOD error of its refinement vertex, see also Section 3.2.

4.2. K-Patch Index Buffers

Each K-patch type of Figure 2 is represented by an array of 2D indices x_i, y_i given on a canonical grid, corresponding to its indexed triangle strip. For example, a 3-patch of type 4 corresponds to the triangle strip with indices $(x_i, y_i) = \{(0,0),(0,1),(1,1),(0,2),(1,1),(1,2),(2,2)\}$. Given K, the necessary eight K-patch triangle strip index arrays are initialized during system start-up. Hence at run-time K-patches can be rendered by sending their indices x_i, y_i as triangle strip coordinates to the GPU and constructing their world coordinates according to (1) in the vertex shader. However, this also requires the shader to perform a texture lookup to get the z value and normal vector from the height-map. For that, the height-map of the M-block must be bound as vertex texture.

To avoid coordinate construction and texture lookups in the vertex shader, index buffers for K-patch triangle strips can explicitly be precomputed, and the M-block height-map is then interpreted as a vertex buffer. An index buffer has to be defined for each possible K-patch type, scale and offset configuration (see also Figure 4(b)). Depending on the ratio of K to M this results in a fixed but possibly large number of explicit index buffers, but it is still independent of the actual height-data and must only be performed once for a certain K and M combination. At run-time, the M-block height-map is thus bound as a VBO and a triangle K-patch is rendered as an indexed triangle strip. In the current implementation and experiments, RASTeR operates with $M = 2K$ and explicit index buffers for K-patches.

5. RASTeR System

To perform fast view-dependent LOD rendering, RASTeR dynamically adapts the selection of triangle K-patches in the meta bintree for each rendered frame. As we outline below, several error factors can be reduced to a single term which gives us the selection factor for a K-patch node. A traversal of the meta binary tree selects all nodes that have to be rendered for the current frame, and their corresponding M-Blocks are activated. In the following, the dynamic run-time behavior of the system is discussed and a diagram of its principal components is given in Figure 6.

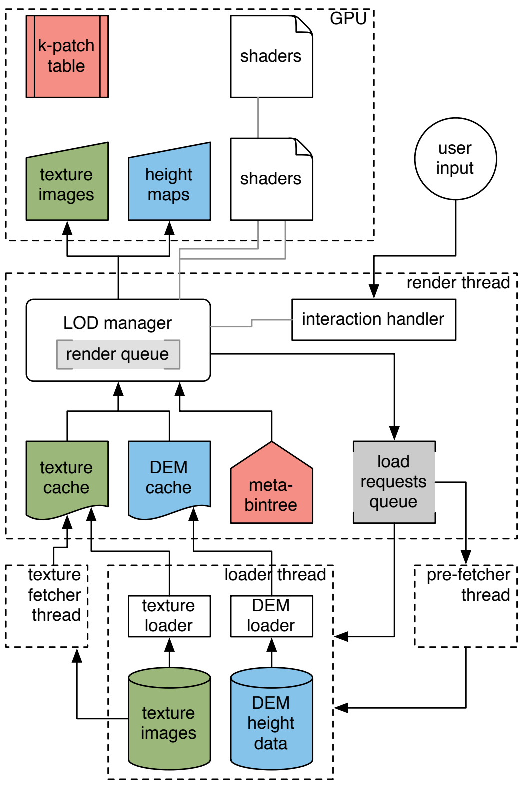


Figure 6: *RASTeR system and resource management.*

5.1. Level-of-Detail Selection

At run-time, the K-patch bintree is traversed by the LOD manager in the rendering thread, and LOD selection is performed on a per K-patch basis. After view-frustum culling, visible K-patch nodes are selected based on their saturated view-dependent octagon error metric [Ger03] to avoid cracks and T-junctions in the terrain surface. Given the focal plane parameter, the object-space error of each K-Patch and the octagon distance, the final error can quickly be computed and compared to a given error tolerance, see also [LP01]. The error tolerance is either given by the user or adjusted to keep an interactive framerate. Traversal stops when the first node with an error below the tolerance is reached. The so selected K-patch nodes activate their corresponding height-field data M-blocks. The system's render queue then renders the associated K-Patches of all active M-blocks.

5.2. Texture Selection

To match the K-patch and M-block structures, textures are managed in square units and organized in a texture mipmap pyramid [Wil83]. Each texture unit can be compressed using GPU compression to almost $1/6$ of its original size. The texture resolution used for one or more related K-patches is chosen depending on the distance from the camera. But to limit excessive use of high-resolution textures, due to triangle patches spanning a wide depth, we additionally constrain the texture mipmap levels such that only a certain percentage is rendered at high(est) resolution near the viewpoint.

5.3. Resource Management

The meta bintree is sufficiently small in size compared to the DEM height field data of the M-blocks since it only stores the saturated octagon error metric parameters and K-patch attributes with each node. Hence it can typically be loaded into main memory, or accessed synchronously (on-demand) via memory mapping from out-of-core. Other K-patch data does not have to be maintained.

Therefore, the only resources which have to be actively managed in the sense of loading, caching and prefetching are the data intensive grid-digital height-field M-blocks and textures. The height-map data of currently activated M-blocks is loaded and stored in GPU memory, and other recently used M-blocks are cached in main memory. GPU memory is freed if M-blocks are not referenced from any K-patches for some time, and height data is unloaded from system memory as well if applicable. In a similar way texture images are dynamically loaded and cached in main and GPU memory as illustrated in Figure 6. Dyanamic loading, caching and prefetching is discussed below in Section 5.5.

5.4. Rendering

During rendering, the system iterates over the render queue of activated M-blocks and their K-patches. For each M-block, several draw-elements calls are issued on its vertex buffer for the selected LOD triangle K-patches associated with it. The M-block's vertex buffer consists only of height (altitude) and surface normal data, since the x and y (long-/latitude) coordinates are computed in the vertex shader according to K-patch type, base vertex and spacing attributes.

During LOD selection, K-patches are grouped by their M-block usage such that the corresponding height-field vertex buffer can be bound once for all corresponding K-patches. The K-patches' index buffers are already available on the GPU as they are static, and simply the index buffers of the selected K-patches must be activated. Given the patch base offset in world coordinates, calculation of the vertex coordinates can be done from the given parameter data.

5.5. Asynchronous and a priori Fetchings

In interactive rendering, the viewing parameters generally vary smoothly over time and thus LOD changes between frames happen gradually and predictively. In case of unavailability of a new LOD node, and to avoid synchronous loading of its M-block data from out-of-core, the LOD change may be delayed to keep the frame rate constant until the new LOD data has been loaded from disk. The requested higher or lower LOD details, which consist of (compressed) height-field M-blocks and their corresponding color textures, are processed asynchronously from a queue by separate threads. We refer to these LOD updates as *asynchronous fetches*.

To support the above strategy we perform incremental frame-to-frame LOD updates as follows. The state of the currently selected and rendered LOD can be viewed as a *front* through the K-patch meta bintree, as indicated in Figure 7. A change in LOD then consists of an incremental update of that front, down- or upwards for LOD refinement or coarsening respectively. With respect to asserting the consistency of the LOD triangulation, incremental K-patch refinement or coarsening is only performed as long as the resolution of the neighboring patches can be matched without introducing cracks. In accordance with the triangulation defined in Section 3.1 neighboring K-patch nodes can thus differ by at most one level.

Hence the actual new updated front constitutes of those nodes from the current and the targeted fronts which are already available in GPU or main memory as well as closest to the targeted front. If the M-block data for targeted nodes is not available then an appropriate request is pushed onto the asynchronous load queue. Moreover, the corresponding nodes from the current front which can thus not be replaced are retained in the render queue instead.

Whenever a K-patch causes an M-block to be pushed onto the load-requests queue, its texture and texture resolution are also considered and fetched if necessary. While uncompressed M-blocks can directly be loaded into memory, compressed formats require on-the-fly decompression.

In the case of a smooth user navigation, asynchronous fetches with delayed LOD display can satisfy almost any targeted updates with little latency. However, abrupt changes in navigation direction, e.g. through sharp rotations, cannot always be handled that way (e.g. node l in Figure 7). Therefore, we make use of the predictive nature of interactive navigation via spatial coherence and prefetch M-blocks that fall

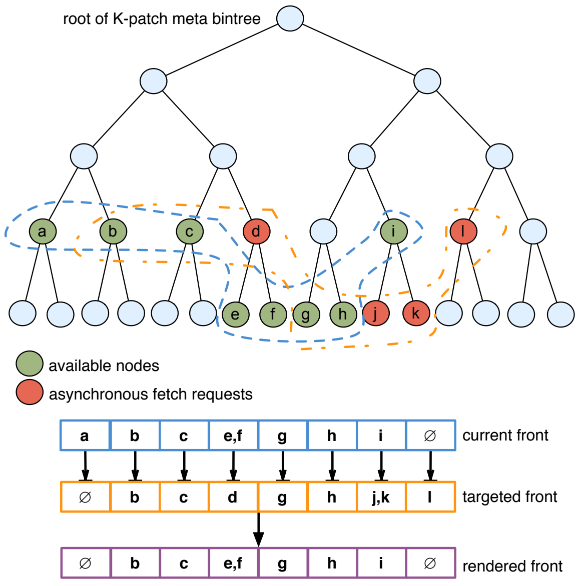

root of K-patch meta bintree
a
b
c
d
i
l
e
f
g
h
j
k
available nodes
asynchronous fetch requests
a | b | c | e,f | g | h | i | ∅
current front
∅ | b | c | d | g | h | j,k | l
targeted front
∅ | b | c | e,f | g | h | i | ∅
rendered front

Figure 7: *Asynchronous fetching.*

within an extended view frustum via the same asynchronous loading request queue.

Eventually, situations may still occur where no existing LOD data can directly be used without introducing cracks , holes and artifacts in the terrain display. In these rare cases a *synchronous fetch* must be executed which loads the required data directly for immediate rendering.

6. Experimental Results

The RASTeR system was implemented for Mac OS X and GNU/Linux in C++, OpenGL and using the OpenGL Shading Language (GLSL) for programmable shaders. We have used an NVIDIA GeForce 8800 GT card and 2.66 GHz Intel Xeon in our experiments. We have tested RASTeR on different DEMs and the results are summarized in Table 1.

In our tests, the height was encoded as 16bit unsigned integer, and the normal was packed into a 16bit unsigned integer as well, using 6 bits for the x, y and 4 bits for the z components. As long as the M-block to K-patch size ratio is small, i.e. 2, a single index buffer can be used to store all indices for all explicit different configurations of K-patches within an M-block. 16bit unsigned integers can be used to encode all the indices in that case. Using 129×129 sized M-blocks and $(K = 65)$-patches has shown to be a good choice for the size of the two basic DEM data and triangulation units.

As demonstrated in Table 1, RASTeR achieves excellent performance for interactive renderings as shown in Figures 8. In Figure 9(a) we present a detailed performance analysis, obtained from rendering the Puget Sound data set. As can be seen, the interactivity and triangle rendering throughput are excellent, achieving a sustained rate of about 200M triangles or 100 frames per second, also compared to previous state-of-the-art methods such as [GMC*06]. Figure 9(b) shows the corresponding data complexity. Note that each K-patch corresponds to an entire triangle strip being rendered.

Figure 9(c) shows the low overhead, that is the ratio of the LOD data structure traversal and loading cost as percentage of to the overall rendering time, which is typically below 10%. Finally, Figure 9(d) strongly supports our system design by demonstrating that the asynchronous DEM data fetches, which are even generally low, prevail over the very few synchronous fetches necessary for unexpected and abrupt view changes.

Data Set	Resolution	Texture	Uncompressed		Compressed	
		GB	Fps	MTps	Fps	MTps
Ofenpass	4K x 3K	0.111	109	251	105	246
Zurich	2K x 2K	3.38	131	241	130	237
Puget Sound	4K x 4K	0.7328	113	249	109	238
Puget Sound	16K x 16k	0.6961	98	223	95	218

Table 1: *Rendering performance for pixel error $= 2$ in frames (Fps) and million triangles per second (MTps). Texture size given after processing and compression.*

7. Conclusion

In this paper we have demonstrated that a simple data layout and patch-based multiresolution triangulation model can achieve very high rendering performance. The benefits of the proposed solution are the separation of the triangulation model from the actual DEM height-field data management. This allows an efficient, multi-scale and regular block based representation of the elevation data in conjunction with an adaptive patch-based bintree triangulation model, that achieves the desired features outlined in the introduction.

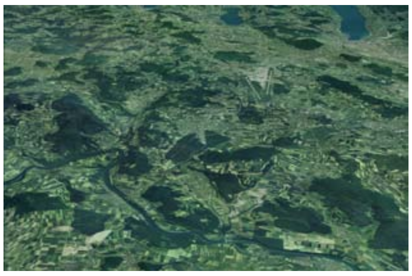
Future work includes evaluations of the sweet spot for the K-patch and M-block size parameters as well as the analysis of possible aliasing problems due to mapping of dense triangles onto a limited number of pixels on screen.

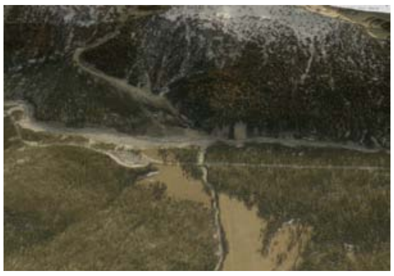
Acknowledgements

We would like to thank the Remote Sensing Lab of the University of Zurich for the Ofenpass DEM data of the Swiss National Park, as well as the Large Geometric Models Archive at GeorgiaTech for the Puget Sound data set. This work was partially supported by the Swiss National Science Foundation Grant 200021-111746/1.

References

[CGG*03] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.: BDAM - batched dynamic adaptive meshes for high performance terrain visualization. In *Proceedings EUROGRAPHICS* (2003), pp. 505–514. also in Computer Graphics Forum 22(3).


(a) Zurich (DHM25 © swisstopo)


(b) Ofenpass (© RSL Univ. of Zurich)

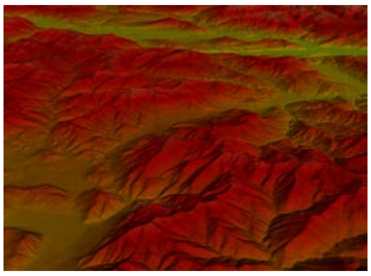
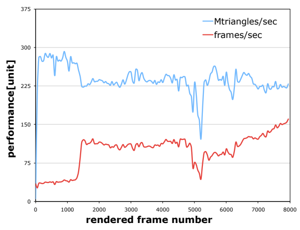
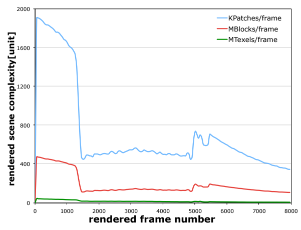

(c) Puget Sound

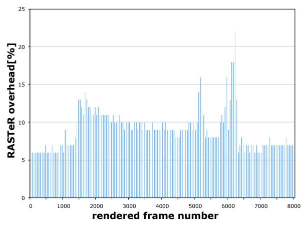
Figure 8: *Example screenshots of interactive terrain rendering of different DEMs.*



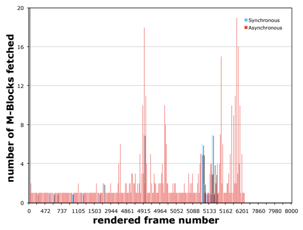
(a)



(b)



(c)



(d)

Figure 9: *(a) Rendering speed. (b) Rendering complexity. (c) RASTeR overhead. (d) Asynchronous vs. synchronous fetches.*

[DFMP96] De Floriani L., Marzano P., Puppo E.: Multiresolution models for topographic surface description. *The Visual Computer 12*, 7 (August 1996), 317–345.

[DWS*97] Duchaineau M., Wolinsky M., Sigeti D. E., Miller M. C., Aldrich C., Mineev-Weinstein M. B.: ROAMing terrain: Real-time optimally adapting meshes. In *Proceedings IEEE Visualization* (1997), Computer Society Press, pp. 81–88.

[Ger99] Gerstner T.: *Multiresolution Compression and Visualization of Global Topographic Data*. Tech. Rep. 29, Institut für Angewandte Mathematik, Universität Bonn, 1999. to appear in Geoinformatica 2001.

[Ger03] Gerstner T.: *Top-Down View-Dependent Terrain Triangulation using the Octagon Metric*. Tech. rep., Institute of Applied Mathematics, University of Bonn, 2003.

[GMC*06] Gobbetti E., Marton F., Cignoni P., Benedetto M. D., Ganovelli F.: C-BDAM – compressed batched dynamic adaptive meshes for terrain rendering. *Computer Graphics Forum 25*, 3 (September 2006), 333–342.

[HDJ05] Hwa L. M., Duchaineau M. A., Joy K. I.: Real-time optimal adaptation for planetary geometry and texture: 4-8 tile hierarchies. *IEEE Transactions on Visualization and Computer Graphics 11*, 4 (2005), 355–368.

[Hop98] Hoppe H.: Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings IEEE Visualization* (1998), Computer Society Press, pp. 35–42.

[Lev02] Levenberg J.: Fast view-dependent level-of-detail rendering using cached geometry. In *Proceedings IEEE Visualization* (2002), Computer Society Press, pp. 259–266.

[LKR*96] Lindstrom P., Koller D., Ribarsky W., Hodges L. F., Faust N., Turner G. A.: Real-time, continuous level of detail rendering of height fields. In *Proceedings ACM SIGGRAPH* (1996), pp. 109–118.

[LP01] Lindstrom P., Pascucci V.: Visualization of large terrains made easy. In *Proceedings IEEE Visualization* (2001), Computer Society Press, pp. 363–370.

[LPT03] Lario R., Pajarola R., Tirado F.: Hyperblock-QuadTIN: Hyper-block quadtree based triangulated irregular networks. In *Proceedings IASTED Invernational Conference on Visualization, Imaging and Image Processing (VIIP)* (2003), pp. 733–738.

[LRC*03] Luebke D., Reddy M., Cohen J. D., Varshney A., Watson B., Huebner R.: *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers, San Francisco, California, 2003.

[OR99] Ohlberger M., Rumpf M.: Adaptive projection operators in multiresolution scientific visualization. *IEEE Transactions on Visualization and Computer Graphics 5*, 1 (January-March 1999), 74–93.

[Paj98] Pajarola R.: Large scale terrain visualization using the restricted quadtree triangulation. In *Proceedings IEEE Visualization* (1998), pp. 19–26,515.

[PG07] Pajarola R., Gobbetti E.: Survey on semi-regular multiresolution models for interactive terrain rendering. *The Visual Computer 23*, 8 (2007), 583–605.

[Pom00] Pomeranz A. A.: *ROAM Using Surface Triangle Clusters (RUSTiC)*. Master's thesis, University of California at Davis, 2000.

[Pup96] Puppo E.: Variable resolution terrain surfaces. In *Proceedings of the 8th Canadian Conference on Computational Geometry* (1996), pp. 202–210.

[SS92] Sivan R., Samet H.: Algorithms for constructing quadtree surface maps. In *Proceedings 5th International Symposium on Spatial Data Handling* (August 1992), pp. 361–370.

[Wil83] Williams L.: Pyramidal parametrics. In *Proceedings ACM SIGGRAPH* (1983), ACM SIGGRAPH, pp. 1–11.