GPU-optimized efficient quad-tree based progressive multiresolution model for interactive large scale terrain rendering

Egor Yusov, Vadim Turlapov
Department of Computational Mathematics and Cybernetics
Nizhny Novgorod State University, Nizhny Novgorod, Russia
yusov_egor@mail.ru, vadim.turlapov@cs.vmk.unn.ru

Abstract

This paper presents new effective method for real-time terrain triangulation and visualization. The method utilizes compact progressive terrain representation which is constructed at preprocess stage and requires a very low memory expense. This representation allows easily LOD extraction at runtime. In contrast to previous works our quadtree-based triangulation is not restricted such that neighboring regions must differ by at most one level in LOD hierarchy. This allows constructing more adaptive triangulation. The method performs LOD selection on per-block basis and thus it is not CPU-consuming and fully exploits power of modern graphics hardware. Patch triangulations are constructed only once the first time the patch is needed and cached in fast video memory in optimized form in order to achieve highest rendering performance. The method is driven by the user-defined screen space error and provides guaranteed surface ε-approximation. The algorithm supports morphing of both geometry and texture (the last is not usually discussed) which hides popping artifacts related with LOD change. We also discuss some system-level aspects such as multithreaded asynchronous implementation of the proposed algorithm, video-memory allocation control, camera position prediction and speculative data prefetch. The proposed method can be easily extended to allow terrain compression.

Keywords: *interactive terrain visualization, multiresolution modeling, quadtree, level-of-detail control, adaptive triangulation, geomorphs.*

1. INTRODUCTION

High-quality interactive large-scale terrain visualization is an important aspect of many applications such as geoinformation systems (GIS), landscape editors, virtual environments, flight or drive simulators, outdoor computer games, etc. Despite the fact that performance of modern graphics processors (GPUs) nowadays reached 10^9 triangles per second and continues to increase, high accuracy data sets still exceed processing capabilities of even the highest-end graphics platforms. Today satellite landscape scans can cover large area with very high resolution (one meter spacing or less) and contain billions of elevation samples. The size of digital terrain data will continue to rise and the gap between rendering capacity of graphics cards and complexity of complete regular terrain model is unlikely to ever disappear. Furthermore, rendering uniformly dense grid can lead to aliasing artifacts caused by many-to-one texel to pixel mappings. Therefore to achieve high image quality at interactive frame rates it is necessary to reduce model complexity as far as it is possible without leading to inferior visual representation.

Ideally, simplification algorithm adapts terrain tessellation based on the *screen-space geometric error*, the deviation in pixels between the approximated surface and the original terrain. The screen-space error is defined by the 1) distance to camera, 2) local surface characteristics, and 3) surface orientation. Constructing of adaptive triangulation is performed on the CPU. It is quite feasible to create simplified mesh which provides very accurate approximation of a particular terrain and consists of a few faces in order to greatly decrease GPU load. However it will consume too much CPU time and such method will be CPU-bound. From the other hand if LOD selection is not used, the triangulation will consist of huge number of primitives, and the method will become GPU-bound. The main challenge is to find the trade-off between these two extreme cases.

2. RELATED WORK

A number of different approaches have been developed during last years to adaptively control terrain tessellation as the function of terrain characteristics and view parameters. These methods can be organized into the following groups:

Triangulated irregular networks (TINs). TIN is an unrestricted triangulation of arbitrary set of vertices. TIN methods can be used to simplify any mesh, and they do not take advantages of high regularity of grid-digital terrain representation. Some such algorithms are based on the principle of 2D Delaunay triangulation [2], [6]. Others allow arbitrary connectivity [5], [8]. For instance terrain rendering method based on progressive application of vertex-split and edge-collapse mesh refinement and simplification operations is presented in [8].

TIN methods provide the best possible surface approximation for a given polygon count. However such methods are too computationally expensive since they require tracking of mesh adjacencies and refinement dependencies. Furthermore, such algorithms consume a lot of memory for internal data structures. While TIN approaches are very efficient for general mesh simplification they are hardly suitable for large scale terrain rendering since such methods are found to be completely CPU-bound.

Quadtree and bintree based methods. Quadtree and bintree based hierarchical terrain triangulation and visualization approaches fully take advantages of high regularity of height field data. In contrast to TIN methods such methods use compact representation and allow fast construction and rendering of adaptive crack-free terrain triangulation. A good survey of different quadtree and bintree based algorithms is presented in [14].

One of the first effective quadtree-based terrain simplification methods is presented in [4]. The method uses screen-space error metric to adaptively control LOD of different surface regions. To create a *matching* triangulation (i.e. triangulation without cracks) the method implies a restriction on the quadtree such that neighboring regions must differ by at most one level in the hierarchy. This restriction defines hierarchical dependency relations between vertices which must be kept in order to guarantee matching triangulation. Famous ROAM algorithm which is conceptually very similar to one proposed in [4] is presented in [7]. ROAM is based on the notion of a triangle bintree hierarchy. An effective screen-space distortion error metric as well as fast priority queue driven terrain triangulation algorithm are presented in [7]. There were a lot of further improvements on these two fundamental methods ([9], [10], [16]).

Alternative terrain simplification approaches which are based on wavelet decomposition are proposed in [3], [23]. In these methods adaptive mesh is constructed based on the significance of wavelet coefficients obtained by applying wavelet transform to the source height field.

All listed here methods were very efficient 5 years ago, but nowadays GPUs greatly outperform CPUs, so constructing adaptive triangulation each frame which involve random memory accesses and transferring data from main memory to local video memory consumes most time while GPU stays practically idle.

Hierarchies of pre-computed geometry blocks (clusters, batches, aggregates etc.). On modern graphics platforms the time that is saved by rendering fewer triangles due to adaptive retriangulation is completely amortized by the time needed to perform the retriangulation. To fully exploit power of latest GPUs recent methods use complex primitives composed of many triangles as minimal element for mesh construction. This speeds up LOD selection stage but generates triangulation that is more redundant than one provided by quadtree or bintree based algorithms listed above. Nevertheless modern GPUs successfully cope with complex scenes and overall performance of the visualization system dramatically increases.

The first terrain rendering method fully exploiting power of the latest GPUs is RUSRiC and is presented in [11]. The main idea of RUSTiC is to replace single triangles in bintree hierarchy of ROAM algorithm from [7] with precomputed triangle clusters composed of many triangles. This idea was further developed in [15] where it was proposed to cache such clusters (called in [15] aggregate triangles) in fast video memory for efficient rendering. Extensions of quadtree-based methods which utilize hyper block as minimal simplification element are presented in [20] and [25]. The same idea as in [11] and [15] is exploited in [18] where single triangles from bintree hierarchy are replaced with small TINs called *batches*. The batches for all resolution levels are constructed off-line and stored on disk in highly optimized for rendering form. In [19] the authors extended their original algorithm to successfully render planet-size terrains at interactive frame rates. However precomputed triangulations consume a lot of storage and require frequent disk access at runtime.

View-independent approaches. In [21] it was proposed to fully refuse view-dependent refinement and instead develop a framework which optimally feeds graphics pipeline. This decision was inspired by the fact that rendering throughput has reached a level that allows covering framebuffer with pixel-sized triangles at video rates. Authors propose the *geometry clipmap* technique which caches terrain in a set of regular nested grids centered about the viewer. Terrain geometry thus depends only on camera position and does not take into account local surface characteristics. A GPU – based geometry clipmap implementation allowed by geometry textures and shader model 3.0 is presented in [22]. Proposed framework provides a lot of benefits. The most important is terrain data compression considered in literature for the first time. Among others advantages are optimal rendering throughput, steady frame rates, terrain details synthesis, simplicity and others.

However the main strength of the proposed algorithm is its main weakness. In order to achieve high visual accuracy it is necessary to render a huge number of polygons. Display resolutions continue to rise and maintaining acceptable quality on large screens would require too many triangles to be rendered. This problem can become more important if advanced terrain shading techniques is used which require a lot of computations and complex environment objects are also rendered.

3. ALGORITHM DESCRIPTION

3.1 Multiresolution terrain representation

The method proposed in this paper is based on one presented in [23]. The initial raw data our algorithm works with is the regular height field, the most commonly used way to define terrain.

At preprocess stage we construct multiresolution representation of the terrain by filtering it into multi-layer pyramid in a certain sense similar to [21]. The layers are numbered from 0 (the coarsest level) to $D_{hf}-1$ (the finest level) where D_{hf} denotes number of levels in the pyramid (or depth of the height field quadtree). The finest level $D_{hf}-1$ is exactly original regular height field. All other layers from $D_{hf}-2$ to 0 have four times lower resolution than the underling level and are constructed by filtering it (figure 1.a). Note that all layers cover the same area but with diminishing accuracy. Vertices in each level are identified by triple index (*i,j,k*), where $k \in \{0,1,...,D_{hf}-1\}$ denotes resolution level and $i,j \in \{0,1,...,2^k-1\}$ denote vertex position in the layer's grid.

This pyramid can also be thought of as a vertices quadtree having D_{hf} depth where each vertex in level *k* has four children in finer level *k*+1 (figure 1.b).

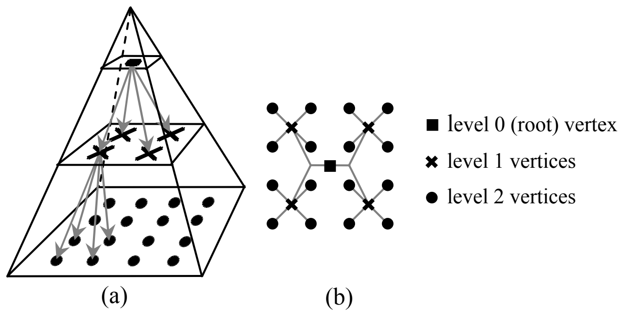

Figure 1: (a) Multiresolution pyramidal representation of 4x4 height field; (b) Corresponding vertices quadtree.

To coarsen each layer we use order 4 Neville interpolating filter [12] which is used for the same purpose in [24]. Weight coefficients of the filter are shown in figure 2. To initialize vertices adjacent to height field borders we extend height field by one sample outside of each border. Height values of additional vertices are taken from the nearest sample of the original height field. So the height of the vertex with *(i,j,k)* indices can be calculated by the following formula:

$$H_{i,j,k} = \sum_{m=2i-1}^{2i+2} \sum_{n=2j-1}^{2j+2} C_{m,n} H_{m,n,k+1}$$

where $H_{m,n,k}$ is the height of (*m*,*n*,*k*) vertex and weight coefficients $C_{m,n}$ are given by the figure 2.

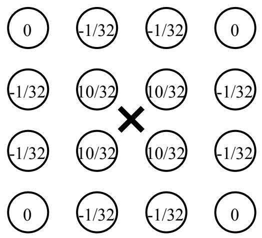

Figure 2: Weight coefficients of the interpolating filter.

3.2 Meta quadtree

As it was discussed in section 2 modern GPUs greatly outperforms CPUs and algorithms like presented in [3, 4, 7, 9, 10, 16, 23] providing fine approximation are completely CPU-bound. Recent approaches [15, 20, 18, 25] are based on per-block simplification. Following these methods we also use terrain patch as minimal element for LOD selection. At preprocess stage we construct collection of patches with different LODs which are organized into quadtree of patches; we call it *meta quadtree* (figure 3). Finest resolution grid of each patch has fixed dimension of $2^{D_p-1}\times 2^{D_p-1}$ vertices. We denote D_p as a *patch depth*. Note that patches cover increasing areas of original terrain with diminishing resolution. If the source height field has $2^{D_{hf}-1}\times 2^{D_{hf}-1}$ dimension, where D_{hf} is the number of levels in the multiresolution pyramid, then the meta quadtree has $D_m = D_{hf} - D_p + 1$ levels (see figure 3). Each patch in the meta quadtree like any vertex in the vertices quadtree can be identified by the triple index (*p*,*q*,*t*) where $t \in \{0,1,...D_m - 1\}$ denotes meta level and $p,q \in \{0,1,...2^t - 1\}$ are patch indices in the meta level.

Triangulation of each patch is built on a subset of vertices from the vertices quadtree. These subsets can be thought of as sub-pyramids in the whole multiresolution pyramidal representation and also can be treated as sub-quadtrees (see figure 3).

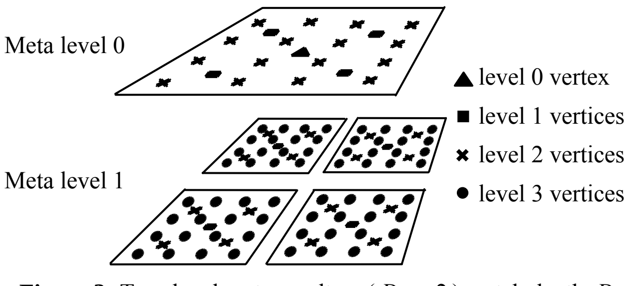

Figure 3: Two-level meta quadtree ($D_m = 2$), patch depth D_p is 3, vertices quadtree depth D_{hf} is 4. Patches of meta level 1 are built on level 3 to level 1 vertices; patch of meta level 0 is built on level 2 to level 0 vertices.

3.3 Patch triangulation

Initial maximum resolution grid corresponds to the full balanced quadtree. Each patch is built on its own sub-quadtree. At patch construction stage we discard some vertices of patch sub-quadtree as well as subtrees grown from these rejected vertices based on surface characteristics. Following the wavelet terminology we denote such discarded trees as *zero-subtrees* [1, 17]. Thus adaptive patch triangulation is given by unbalanced quadtree. The remaining set of vertices we denote as *active*. The density of active vertices should be adaptive to surface characteristics – it should be high in sharp terrain regions and low in smooth areas. We will discuss later in this paper how to define set of active vertices. In contrast to all previous adaptive quadtree and bintree triangulations [4, 7, 9, 10, 16] we does not imply any restriction on quadtree. To guarantee matching triangulation algorithms presented in [4, 9, 16] require that neighboring vertices differ by at most one level in LOD hierarchy. The same restriction is applied to triangle bintree in [7, 10]. This restriction limits adaptability of the polygonal approximation. Our triangulation does not have such drawback. It can be built on any set of active vertices of arbitrary unbalanced quadtree.

As it was said earlier the triangulation is built on set of active vertices which are leaves of adaptive unbalanced quadtree. These vertices are also roots of discarded zero-subtrees. The main triangulation rule is the following [17]: if we imagine that in full resolution uniformly dense triangulation all vertices from discarder zero-subtrees gravitate to their roots and all degenerate triangles are then rejected we will get the desired triangulation (this rule is illustrated in figure 4). You can find more information on triangulation construction scheme as well as effective top-down algorithm outline in [23].

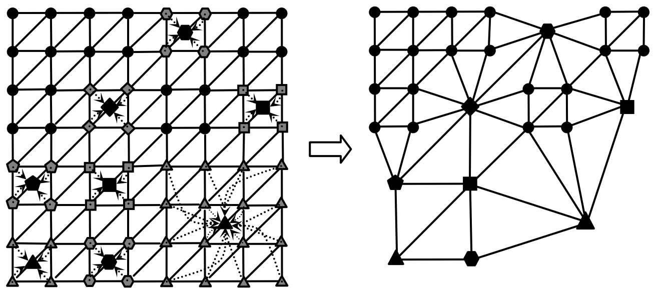

Figure 4: Construction of quadtree triangulation. Active vertices are shown in black, vertices in discarder zero-subtrees are shown in grey, with the same shape as the zero-subtree root vertex they gravitate to.

In [17] as well as in [23] it was not clarified how border vertices are triangulated. In order to solve this problem we insert some additional border vertices in the patch's mesh. Each level of the patch quadtree is extended with one vertex wide border rectangle. Heights of additional vertices are calculated as average height of two (for non-corner vertices) or four (for corner vertices) nearest vertices from the same level (see figure 5.a). To triangulate patch border we connect border-adjacent vertices with the same level nearest border vertex using the same triangulation rule as for the patch interior (figure 5.b).

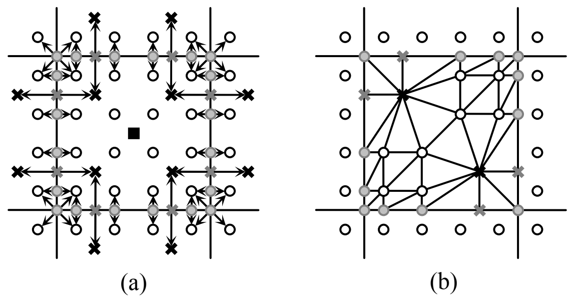

Figure 5: (a) Extending patch mesh with border vertices (shown in grey). Arrows from each border vertex show vertices from the original height field which are used to calculate its height. (b) Patch triangulation with border vertices.

Note that the border vertices are added only when patch mesh is created at run time and calculated on the fly, but not stored permanently. In reality border vertices comprise no more than 15% percents of total mesh size.

The main challenge related with block-based terrain partitioning is eliminating cracks between patches. In order to address this problem we use vertical flanges around patch boundaries, introduced in [13]. Advantage of this method is that all patches can be treated absolutely independently one from another and there is no necessity to imply any restrictions on patch meta quadtree. Another useful for us property is that we can calculate for each patch its world space error. If we did not use border vertices (as in [23]) then patch triangulation as well as this error would depend on LOD of neighbor patches. The drawback of this approach is slightly increase of mesh complexity.

3.4 Multiple LOD construction

In our previous work [23] we select vertices based on the significance of their wavelet coefficients. This approach was not able to provide guaranteed ε-approximation. In this method we eliminate this shortcoming. At preprocess stage we perform adaptive triangulation of the patches and create compact progressive terrain representation which stores information about LOD changes. The construction algorithm is described below.

The finest patch approximation is given by the full balanced quadtree (figure 6.a). To coarsen patch we discard 4 sibling vertices and replace them with their parent (figure 6.b).

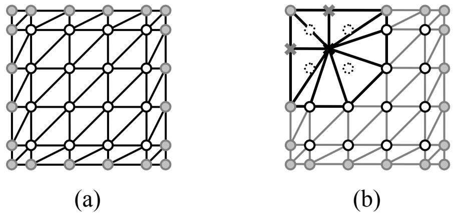

Figure 6: (a) Full patch triangulation. (b) Coarsened triangulation. Discarded vertices are showed by dotted line, changed region of the mesh is shown by thick black lines, and the unchanged region is shown in grey.

We call vertex *mergible* if all its 4 children are active vertices and are already present in partially constructed approximation. All finest resolution vertices are mergible by definition. At construction process we identify all mergible vertices from finest to coarsest level and merge them by discarding 4 children if it does not introduce intolerable error. For the patches from the finest meta level we use threshold τ defined by the user in world space (in meters, inches, etc.). Thus the finest terrain approximation does not introduce world-space error greater than τ (which can be set to 0 to preserve smallest geometrical details). LOD hierarchy construction algorithm can be more specifically defined in the following manner.

Let $\mathrm{proj}_{xy}(t)$ be the projection of triangle *t* on the XY-plane. We define covering $\mathrm{cov}(t)$ of triangle *t* as the set of all vertices at finest resolution whose projection lies in the $\mathrm{proj}_{xy}(t)$:

$\mathrm{cov}(t) = \{v \in V_{D_{hf}-1} : \mathrm{proj}_{xy}(v) \in \mathrm{proj}_{xy}(t)\}$ where $V_{D_{hf}-1}$ is the set of finest resolution vertices (at level $D_{hf}-1$). We define error of triangle *t* as the maximum deviation of all vertices from the triangle covering to triangle's plane:

$Err(t) = \max_{v \in \mathrm{cov}(t)} d(t,v)$ where *d(t,v)* is the distance from the vertex *v* to triangle's *t* surface.

Finally we define error of triangulation *T* as maximum error of all triangles it consists of:

$$Err(T) = \max_{t \in T} Err(t)$$

Let V_T be the set of active vertices of partially constructed approximation and T be the corresponding triangulation. We are considering some mergible vertex *v*. Discarding 4 its children w_1, w_2, w_3 and w_4 can change active vertices set and triangulation in the following way:

$V'_T = V_T / \{w_1, w_2, w_3, w_4\} \cup \{v\}$; $T \to T'$ (see figure 6 as example)

We accept this change if $Err(T') < \tau$. Otherwise we reject this change and move to the next mergible vertex. Note that border vertices are auxiliary and they are not considered to calculate triangulation error.

To examine inequality $Err(T') < \tau$ there is no need to check all triangles in T'. Since $Err(T) < \tau$ (due to construction rule), the intolerable error can be introduced only by new triangles (such triangles are depicted in black in figure 6.b) in the changed area of the mesh. We denote these triangles as ΔT : $\Delta T = \{t \in T' : t \notin T\}$. So to accept merging it is enough to examine the following inequality: $Err(\Delta T) < \tau$.

Patch construction process starts from the full resolution mesh only for the patches at the finest meta level ($D_m - 1$). Construction of patches at meta levels from $D_m - 2$ to 0 starts from some initial triangulation which is obtained by copying the sets of active vertices of child patches. If child patch contains vertices at finer resolution, which are unavailable for the parent patch, then all these vertices are discarded and replaced with their parent vertices which present in parent patch sub quadtree. The threshold for these patches is defined by the initial approximation given by child patches and by rejecting finest-level vertices.

The above algorithm can be summarized as follows. All patches of meta quad tree are processed independently bottom up. Triangulation of patches at finest meta level starts from the full resolution mesh and construction of patches at other levels starts from triangulation given by copying the sets of active vertices of child patches. For all mergible vertices equation $Err(\Delta T) < \tau$ is checked and if it is satisfied, the mesh is coarsened. Threshold τ for the finest meta level is defined by the user and for all other levels it is computed from child triangulations. Computed world space errors are stored in a quadtree data structure and are used at run time to select appropriate LODs.

Note that for patches from coarse meta levels calculating $Err(\Delta T)$ as described above can be computationally expensive because coverings of triangles from ΔT can contain a lot of vertices. To simplify calculation this error can be approximated as the sum of maximum error of child patches and distance from the removed vertices to the polygonal approximation given by the child patches.

Key property of the proposed construction algorithm is that the set of active vertices of parent patch is entirely included into active vertices sets of child patches. This allows constructing compact progressive LOD representation.

3.5 Progressive LOD representation

The obvious way to hold pre-computed patches is to store their triangulations. This approach is used for example in [11, 20, 18, 19, 25] but consumes a lot of memory. Another way is to examine vertex errors and to build triangulations on the fly as in [15] but this method requires a lot of computations at run time. We propose compromise solution: at preprocess stage we construct compact progressive representation which helps to build triangulation at run time very fast.

Patch triangulation is fully defined by the set of its active vertices. To refine mesh some active vertices of parent patch have to be split and emerged child patches have to be triangulated. Mesh coarsening is an inverse process. Hence to know what vertices in patch quadtree are active it is enough to know what vertices are active in parent patch and what vertices have to be split (see figure 7).

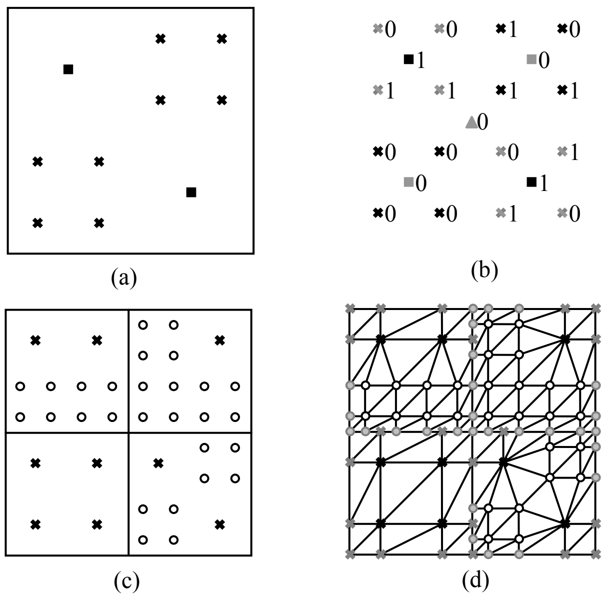

Figure 7: (a) Active vertices of the parent patch quad tree. (b) Vertices that must be split in the parent patch to refine mesh. (c) Active vertices of 4 child patches. (d) Triangulation of child patches.

Thus to progressively encode all LODs from coarsest to finest resolution it is enough to store the set of active vertices of the coarsest patch (the root node of the meta quadtree) and to keep one bit per vertex for each patch sub quadtree to mark which vertices have to be split. Note that this progressive information has to be stored for all patches except for the lowest level of the meta quadtree, because patches at the lowest meta level provide the best approximation and there is nowhere to refine the mesh.

Sub quadtree of the patch having depth D_p has no more than $N_V = 4/3\cdot(2^{D_p-1}\times 2^{D_p-1})$ vertices. Thus progressive information of one patch consumes $N_V/8$ bytes. Number of patches in meta quadtree having D_m levels excluding finest resolution level is $N_P = 4/3\cdot(2^{D_m-2}\times 2^{D_m-2}) = 1/3\cdot(2^{D_m-1}\times 2^{D_m-1})$

Hence total amount of memory required to keep progressive data is $M = N_P N_V/8 = (2^{D_p+D_m-2}\times 2^{D_p+D_m-2})/18$

Since $D_m = D_{hf} - D_p + 1$, M can be calculated as follows:

$M = (2^{D_{hf}-1}\times 2^{D_{hf}-1})/18$ where $2^{D_{hf}-1}\times 2^{D_{hf}-1}$ is the size of the source regular height field. Thus to store the whole LOD hierarchy our method consumes approximately 0.06 bytes per sample of original height field.

Note that we use a very simple method to encode hierarchy which requires 1 bit per vertex. However more complicated techniques based on recursive quadtree traversal can significantly reduce this number.

3.6 Geometry and texture morphing

During interactive fly-over of a large scale terrain special attention has to be paid to maintain temporal continuity of the rendered adapted mesh. When detail level of a particular terrain region is switched due to camera motion, geometry and texture of that region is suddenly changed. This artifact makes annoying impression and called *popping*. In order to solve this problem some approaches propose to use a very little screen-space error threshold [15, 18, 19, 24] so that popping artifacts become unnoticeable. However on large screens this can lead to necessity to render a huge number of primitives. Another approaches use so called *geomorphs* [25] to smoothly change LOD. A problem usually left uncovered is supporting not only geometry, but also texture morphing.

To shade terrain surface each patch is usually assigned one or more textures (in our case we apply normal map to each patch). Resolution of the textures is defined by the patch LOD. The issue which is usually left untouched is correct connection of textures at different LOD. If no special actions are taken, texture junctions are noticeable (figure 8.a). Below we propose a solution to all mentioned problems.

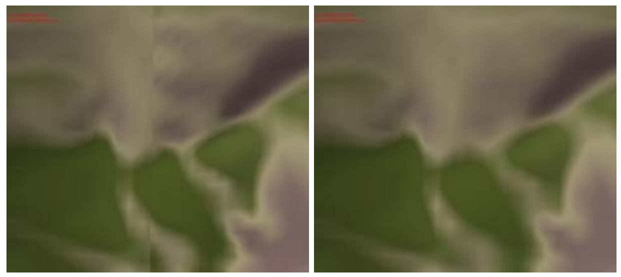

Figure 8: Left: sharp edge on junction of patches with different LOD. Right: eliminating sharp edge by texture blending.

To eliminate popping artifacts we use morphing between successive LODs for both geometry and texture. Geometry morphing (geomorph) is performed for the whole patch and is controlled by the morph parameter $\alpha \in [0,1]$. 0 corresponds to the patch own resolution level and 1 corresponds to next coarser level. When patch is about to be coarsened, the α value begins to ramp from 0 to 1 and reaches 1 exactly when the patch is coarsened. When patch is refined, emerged child patches have morph parameters equal to 1 and therefore they match parent patch geometry and texture. When camera approaches these patches, morph parameter smoothly falls to 0 and patches obtain their own LOD.

The geomorph is performed in much the same way as it was proposed in our previous work [23]. The main distinction is that difference in LOD of the vertex and its parent can be more than one level. The geomorph is performed in a vertex shader. When morph value is greater than 0, all vertices of the patch begin to move to their parents, defined by the parent patch (figure 9). In our current implementation each vertex stores two additional values necessary to perform morphing: the first value is the difference between detail level of the vertex and its parent. This number lies in range $[0, D_p - 1]$. The second value is the height of the vertex parent. Note that vertex textures and shader model 3.0 allow vertex heights to be stored in a vertex texture which can be accessed by the vertex shader. In this case it is enough to store only LOD change value which together with the vertex indices can be used to calculate parent indices and retrieve its height from the texture.

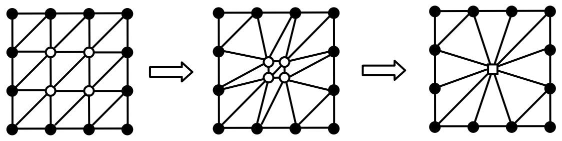

Figure 9: Performing geomorph.

In order to both provide texture morphing when LOD is about to change and seamlessly connect patches with different resolutions we store two normal maps for each patch: the fine normal map and the normal map corresponding to next coarser level (the same approach can be used in case of any other type of texture, for instance, photo texture). In order to implement texture blending we use four values and special four-channel blend weight texture. For each patch we calculate relative blend factors $\tilde{\alpha}_l$, $\tilde{\alpha}_r$, $\tilde{\alpha}_b$, and $\tilde{\alpha}_t$ of left, right, bottom and top neighbor patches respectively which represent LOD of corresponding neighbor patch relative to LOD of current patch. We store these values as pixel shader constants. These relative blend factors are determined as follows: $\tilde{\alpha}_n = \max(\alpha, \alpha_n + d),\ n = l, r, b, t$ where d is the difference between LOD of current and neighbor patch ($d < 0$ if the neighbor patch has finer resolution and $d > 0$ if the neighbor patch has coarser resolution); α_l, α_r, α_b, and α_t are morph values of the neighbor patches. When two patches with different LOD have shared border, the patch with the finer resolution is responsible for correct texture connection. We use special texture which defines which fraction of neighbor patch blend factor should be used for particular pixel. This texture is depicted on figure 10.

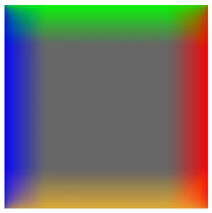

Figure 10: Neighbor patch LOD weights texture. Blue color represents weight of the left neighbor's blend value $\tilde{\alpha}_l$, green – weight of the top neighbor's blend value $\tilde{\alpha}_t$, red and orange – weights of right and bottom neighbor's blend values $\tilde{\alpha}_r$ and $\tilde{\alpha}_t$; grey color represents weight of patch own morph value α.

This texture assures that texture LOD of the current patch smoothly changes and that at patch border it matches texture LOD of the neighbor patch. To calculate blend value for particular pixel we use the following equation:

$$L_{blend} = \tilde{\alpha}_l w_l + \tilde{\alpha}_r w_r + \tilde{\alpha}_b w_b + \tilde{\alpha}_t w_t + \alpha(1 - (w_l + w_r + w_b + w_t)) =$$
$$= (\tilde{\alpha}_l, \tilde{\alpha}_r, \tilde{\alpha}_b, \tilde{\alpha}_t) \bullet (w_l, w_r, w_b, w_t) + \alpha(1 - (w_l, w_r, w_b, w_t) \bullet (1,1,1,1))$$

where w_l, w_r, w_b, w_t are weights of left, right, bottom and top neighbor blend value correspondingly. These four values are stored in red, green, blue and alpha components of the texture. The mentioned equation guarantees that on patch border LOD exactly matches the LOD of the corresponding neighbor patch. To calculate resulting color of particular pixel we use the following formula:

$c = c_f(1 - L_{blend}) + c_c L_{blend}$ where c_f is the color defined by the fine texture and c_c is the color given by a coarse one. Note that this scheme works well when resolution of adjacent patches differ at most by one. However in practice this is the prevalent case.

3.7 Run time LOD selection

For all patches in meta quad tree except for the patches from the lowest meta level we store minimum and maximum height values. These values are necessary to compute patch bounding box extents. Note that there is no need to keep $x_{\min}$, $x_{\max}$, $y_{\min}$ and $y_{\max}$ values since they can be computed based on patch indices and its meta level. To approximate screen space error of the patch we use the following equation:

$\varepsilon_{scr} = \dfrac{S}{tg(\gamma/2)} \dfrac{\varepsilon}{d}$ where S is the screen resolution (maximum of horizontal and vertical resolutions), γ is the field of view angle, ε is the patch geometric world space error calculated at preprocess stage, and d is the distance from the camera to the bounding box. This formula does not take into account surface orientation because this does not bring significant LOD gain, but substantially complicates computations. Since patch approximations provide guaranteed world space error bound (due to construction scheme), the given formula provides guaranteed screen-space error bound of the patch.

At run time before rendering each frame we recursively traverse meta quadtree and check screen space errors of the patches. During this process we increase LOD for the regions with screen space error greater than user-defined threshold and decrease it where it does not introduce intolerable error. This simple top-down algorithm generates adaptive approximation which satisfies user-defines screen space error threshold. At this stage we also perform view-frustum culling and calculate morph values.

4. IMPLEMENTATION

As it was discussed in Section 2 constructing very accurate approximation each frame and transferring data from main to video memory is not suitable for current graphics platforms. To best exploit power of modern GPUs we cache data of terrain patches in the fast GPU-accelerated video memory and use it across many successive frames. CPU performs meta quad tree traversal and selection of appropriate LOD for different areas of the terrain based on patch geometric world space error and distance to camera. Since LOD selection is carried out on a per-patch basis this operation requires very low time cost. CPU also performs view-frustum culling and computes morph coefficients. When LOD changes CPU builds adaptive triangulation using progressive representation. The triangulation is constructed only once first time the patch is needed and cached in the fast GPU-accelerated memory. Thus slow data transfer between CPU and GPU occurs very rarely (once per multiple frames) only at patch creation time. Effective parallel multi-threaded asynchronous implementation of the algorithm completely hides patch creation delays and smooth frame rates. Geometry morphing is performed by the GPU in the vertex shader on a per-block basis and provides temporal and visual continuity. The shader outline is presented on figure 11.

```
VS_OUTPUT TerrainVS(in float3 IJLevel : POSITION0,
                    in float2 VertAndParentH : TEXCOORD0,
                    in float LODDiff : TEXCOORD1)
{
  VS_OUTPUT Output;
  float3 Pos;
  Pos.xy = ComputeVertexWorldXYCoords(IJLevel.xyz);
  Pos.z  = VertAndParentH.x;
  float3 ParentIJLevel;
  ParentIJLevel.xy = floor( IJLevel.xy / exp2(LODDiff) );
  ParentIJLevel.z = IJLevel.z - LODDiff;
  ParentPos.xy = ComputeVertexWorldXYCoords(ParentIJLevel.xyz);
  ParentPos.z  = VertAndParentH.y;
  Pos.xyz = lerp(Pos, ParentPos, g_PatchMotphCoeff);
  Output.TexUV.xy = Pos.xy / g_PatchExtents.xx + g_TexCoordShift.xx;
  Pos.xy += g_PatchLBCornerXY.xy;
  Output.Pos = mul( float4(Pos, 1), g_mWorldViewProj );
  return Output;
}
```

Figure 11: Listing of the HLSL code of the vertex shader.

Patch data is divided into three parts: the first vertex buffer contains (i,j) indices of the vertex and its level in the patch quad tree. This data is constant for all patches. The second vertex buffer contains height of the vertex and its parent. And the third buffer contains the difference between detail level of the vertex and its parent. These values are used to calculate world coordinates of the vertex as well as coordinates of the vertex parent and morphed position. Since our current implementation is based on vertex and pixel shader model 2.0 all vertex buffers have the same size which is equal to the number of nodes in the patch quad tree. One can see that the height data is duplicated in each vertex. In shader model 3.0 and 4.0 this overhead can be eliminated by storing height data in textures which can be accessed by the vertex shader. This will allow reduce space cost by two times. Since shader model 3.0 does not support integer operations we store LOD difference as float despite the fact that it is far enough to use 8-bit integer to store this value. Introducing DX10 and shader model 4.0 can solve this problem. Besides it is clear that four sibling vertices have the same parent, so LOD difference data texture should have quarter size of the height data texture. Furthermore in DX10 the first vertex buffer can be eliminated at all and vertex indices can be calculated based on vertex ID.

5. RESULTS AND DISCUSSION

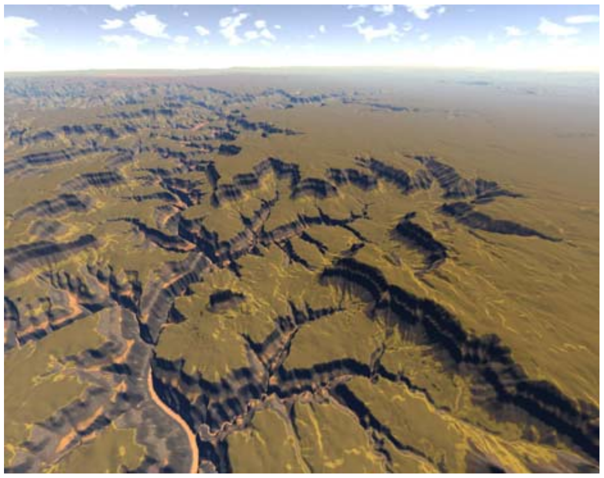

Figure 12: View of the Grand Canyon data set rendered by our algorithm. Terrain color is calculated based on surface height and slope.

Proposed algorithm was implemented in a test terrain visualization system. The code is written in C++ in MS Visual Studio .NET environment. We use DirectX 9.0 as the graphical API.

We tested our system on an 8192x8192 elevation data set. Preprocess stage took a few minutes and produced 3.5 MB of progressive information. For comparison precomputed batches for the same size height field in [18] consume about 1.5 GB of disk space. Of course these batches provide perfect surface approximation, but this is not essential for the current graphic hardware.

During our tests the camera was moving at very high speed, its trajectory included a lot of sharp turns. For the test purposes we used simple height-based texturing since we wanted to evaluate maximum rendering performance. Using more complex coloring (as shown on figure 12) increases only GPU load. We used two machines to test our system. The first is the laptop IBM T43p with the following configuration: Intel Pentium M 1.86GHz CPU, 2GB of RAM, ATI MOBILITY FireGL V3200 GPU with 128 MB of video memory. We used patch size of 128x128 vertices, and 7-level meta quadtree, we rendered to a 1352x1069 window, screen space tolerance was set to 4 pixels. System performance graphs are presented on figure 13.

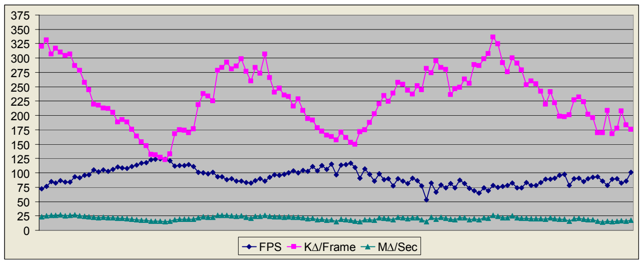

Figure 13: Performance of the system on the first test machine.

This figure shows that our system is suitable even for the platforms with low-end graphics processors. The system operated at stable rendering performance and frame rates never dropped below 50 fps.

Our second test machine is a desktop with dual-core Intel Pentium D 3.4GHz processor, 2GB of RAM, powered by NVIDIA GeForce 7950 GT GPU with 512 MB of local video memory. We used the same patch size 128x128 vertices and 7-level meta quadtree. Window size was 1024x768, screen space tolerance was 1.5 pixels. Acquired performance graphs are shown on figure 14.

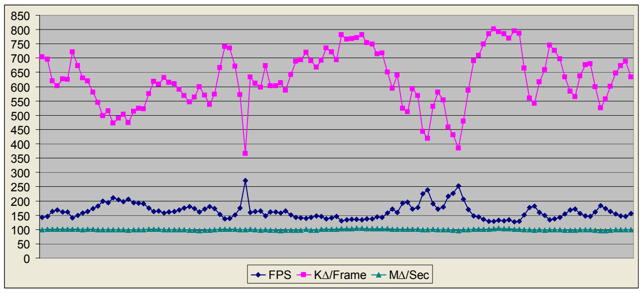

Figure 14: Performance of the system on the second test machine.

The last graphs prove effectiveness of our method and its parallel asynchronous implementation. Since second machine has two processors, one processor is busy with patch creation while the other traverses quadtree and renders the scene. This allows support practically constant rendering throughput at the level of 100 millions triangles per second. The frame rate never dropped below 120 fps which is far enough for real-time terrain visualization.

These tests show that our system is not CPU-limited and highly effective on modern GPUs.

6. CONCLUSION AND FUTURE WORK

We have presented new efficient real-time terrain simplification and visualization algorithm. At preprocess stage the method constructs hierarchy of patches with increasing accuracy and encodes this hierarchy into compact progressive representation. The algorithm uses new approach for building adaptive approximation which allows matching triangulation to be constructed based on any unbalanced quadtree. The algorithm caches patch geometry in fast GPU-accelerated video memory and uses it across many successive frames thus eliminating the need to frequently transfer data from main to video memory and fully exploiting power of modern GPUs. Our system supports morphing of both geometry and texture which is performed in vertex and pixel shader respectively and completely hides popping artifacts. This allows the system to work well even on low-end graphics platforms.

Since uploading data on-demand from secondary storage is not always convenient and induces huge space costs and long reading latencies a few recent methods [21], [25] chosen to utilize terrain compression instead and keep all information in system memory. Our approach is designed specially with intention to be extended to support compression of geographical data. For instance a wavelet transformation can be used to effectively compress multiresolution pyramid we use in our method. DX10 will allow almost all decompression operations to be performed on the GPU.

7. REFERENCES

[1] Shapiro J.M. *Embedded Image Coding Using Zerotrees of Wavelet Coefficients* // IEEE Trans. on Signal Processing. - 1993. - Vol. 41, No. 12. - P. 345-362.

[2] Andreas Voigtmann, Ludger Becker, and Klaus Hinrichs. *Hierarchical surface representations using constrained delaunay triangulations.* In Thomas C. Waugh and Richard G. Healey, editors, Proc. 6th Int. Symposium on Spatial Data Handling, volume 2 of Advances in GIS Research, pages 848-867. Taylor & Francis, London, 1994.

[3] Markus H. Gross, Roger Gatti, and Oliver G. Staadt. *Fast multiresolution surface meshing.* In Proceedings Visualization 95, pages 135-142. IEEE Computer Society Press, 1995.

[4] 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 Proceedings SIGGRAPH 96, pages 109-118. ACM SIGGRAPH, 1996.

[5] De Floriani, L, Magillo, P. and Puppo, E. 1997. *Building and traversing a surface at variable resolution.* IEEE Visualization 1997, 103-110.

[6] Cignoni, P., Puppo, E., Scopigno, R. 1997. *Representation and visualization of terrain surfaces at variable resolution.* The Visual Computer 13(5), 199-217.

[7] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. *Roaming terrain: Real-time optimally adapting meshes.* In Proceedings Visualization 97, pages 81-88. IEEE, Computer Society Press, Los Alamitos, California, 1997.

[8] Hugues Hoppe. *Smooth view-dependent level-ofdetail control and its application to terrain rendering.* In Proceedings Visualization 98, pages 35-42. IEEE, Computer Society Press, Los Alamitos, California, 1998.

[9] Renato Pajarola. *Large scale terrain visualization using the restricted quadtree triangulation.* In Proceedings Visualization 98, pages 19-26 and 515. IEEE Computer Society Press, 1998.

[10] Thomas Gerstner. *Multiresolution visualization and compression of global topographic data.* Technical Report 29, Institut für Angewandte Mathematik, Universität Bonn, 1999. to appear in Geoinformatica.

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

[12] KOVACEVIC J., SWELDENS W.: *Wavelet families of increasing order in arbitrary dimensions.* IEEE Transactions on Image Processing 9, 3 (2000), 480-496.

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

[14] Renato Pajarola. *Overview of quadtree-based terrain triangulation and visualization.* Technical Report UCI-ICS-02-01, I&C Science, University of California Irvine, 2002.

[15] Joshua Levenberg *Fast View-Dependent Level-of-Detail Rendering Using Cached Geometry.* In Proceedings IEEE Visualization'02 (Oct 2002), IEEE, pp. 259-266.

[16] Renato Pajarola, Marc Antonijuan, Roberto Lario. *QuadTIN: Quadtree based Triangulated Irregular Networks.* In Proceedings IEEE Visualization 2002, pages 395-402. IEEE Computer Society Press, 2002

[17] Переберин А.В. *Многомасштабные методы синтеза и анализа изображений.* Диссертация на соискание ученой степени канд. физ.-мат. наук, Институт прикладной математики им. М.В. Келдыша, Москва-2002.

[18] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, R. Scopigno 2003 *BDAM - Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization.* Computer Graphics Forum, Volume 22(Sept. 2003), Number 3, pages 505-514.

[19] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, R. Scopigno 2003 *Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM).* In IEEE Visualization (2003), pp. 147-154.

[20] Roberto Lario, Renato Pajarola, Francisco Tirado 2003 *HyperBlock-QuadTIN: Hyper-Block Quadtree based Triangulated Irregular Networks.* IASTED International Conference on Visualization, Imaging and Image Processing (VIIP 2003), pp. 733-738.

[21] Frank Losasso, Hugues Hoppe *Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids.* ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004) 23(3), pp. 769-776.

[22] Arul Asirvatham, Hugues Hoppe *Terrain Rendering Using GPU-Based Geometry Clipmaps.* GPU Gems 2. Addison-Wesley, 2005, ch. Terrain Rendering Using GPU-Based Geometry Clipmaps, pp. 27-46. http://research.microsoft.com/~hoppe/.

[23] Egor Yusov, Vadim Turlapov *Dynamic terrain simplification based on Haar transform and vertices quadtree.* In Proc. of Conf. on Comp. Graph. and Applications – GraphiCon'2006, Novosibirsk, Russia, June 1 - 5, 2006.

[24] E. Gobbetti, F. Marton, P. Cignoni, M. Di Benedetto, and F. Ganovelli 2006 *C-BDAM - Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering.* Computer Graphics Forum, Volume 25(2006), Number 3

[25] Schneider J, Westermann R 2006 *GPU-Friendly High-Quality Terrain Rendering.* Journal of WSCG ISSN 1213-6972, Vol.14, 2006, Plzen, Czech Republic

About the authors

Egor Yusov is a Ph.D. student at Nizhny Novgorod State University, Department of Computational Mathematics and Cybernetics. His contact email is yusov_egor@mail.ru.

Vadim Turlapov is a professor at Nizhny Novgorod State University, Department of Computational Mathematics and Cybernetics. His contact email is vadim.turlapov@cs.vmk.unn.ru.