High-Performance Terrain Rendering Using Hardware Tessellation

Egor Yusov
Intel Corporation
30 Turgeneva street,
603024, Russia, Nizhny Novgorod
egor.a.yusov@intel.com

Maxim Shevtsov
Intel Corporation
30 Turgeneva street,
603024, Russia, Nizhny Novgorod
maxim.y.shevtsov@intel.com

ABSTRACT
In this paper, we present a new terrain rendering approach, with adaptive triangulation performed entirely on the GPU via tessellation unit available on the DX11-class graphics hardware. The proposed approach avoids encoding of the triangulation topology thus reducing the CPU burden significantly. It also minimizes the data transfer overhead between host and GPU memory, which also improves rendering performance. During the preprocessing, we construct a multiresolution terrain height map representation that is encoded by the robust compression technique enabling direct error control. The technique is efficiently accelerated by the GPU and allows the trade-off between speed and compression performance. At run time, an adaptive triangulation is constructed in two stages: a coarse and a fine-grain one. At the first stage, rendering algorithm selects the coarsest level patches that satisfy the given error threshold. At the second stage, each patch is subdivided into smaller blocks which are then tessellated on the GPU in the way that guarantees seamless triangulation.

Keywords
Terrain rendering, DX11, GPU, adaptive tessellation, compression, level of detail.

1. INTRODUCTION
Despite the rapid advances in the graphics hardware, high geometric fidelity and real-time large scale terrain visualization is still an active research area. The primary reason is that the size and resolution of digital terrain models grow at a significantly higher rate than the graphics hardware can manage. Even the modest height map can easily exceed the available memory of today's highest-end graphics platforms. So it is still important to dynamically control the triangulation complexity and reduce the height map size to fit the hardware limitations and meet real-time constraints.

To effectively render large terrains, a number of dynamic multiresolution approaches as well as data compression techniques have been developed in the last years. These algorithms typically adapt the terrain tessellation using local surface roughness criteria together with the view parameters. This allows dramatic reduction of the model complexity without significant loss of visual accuracy. Brief overview of different terrain rendering approaches is given in the following section. In the previous methods, the adaptive triangulation was usually constructed by the CPU and then transferred to the GPU for rendering. New capabilities of DX11-class graphics hardware enable new approach, when adaptive terrain tessellation is built entirely on the GPU. This reduces the memory storage requirements together with the CPU load. It also reduces the amount of data to be transferred from the main memory to the GPU that again results in a higher rendering performance.

2. RELATED WORK
Many research papers about adaptive view-dependent triangulation construction methods were published in the last years. Refer to a nice survey by R. Pajarola and E. Gobbetti [PG07].

Early approaches construct triangulated irregular networks (TINs). Exploiting progressive meshes for terrain simplification [Hop98] is one specific example. Though TIN-based methods do minimize the amount of triangles to be rendered for a given error bound, they are too computationally and storage demanding. More regular triangulations such as bintree hierarchies [LKR+96, DWS+97] or restricted quad trees [Paj98] are faster and easier to implement for the price of slightly more redundant triangulation.

Recent approaches are based on techniques that fully exploit the power of modern graphics hardware. CABTT algorithm [Lev02] by J. Levenberg as well as BDAM [CGG+03a] and P-BDAM [CGG+03b] methods by Cignoni et al exploit bintree hierarchies of pre-computed triangulations or batches instead of individual triangles. Geometry clipmaps approach [LH04] renders the terrain as a set of nested regular grids centered about the viewer, allowing efficient GPU utilization. The method exploits regular grid pyramid data structure in conjunction with the lossy image compression technique [Mal00] to dramatically reduce the storage requirements. However, the algorithm completely ignores local surface features of the terrain and provides no guarantees for the error bound, which becomes especially apparent on high-variation terrains.

Next, C-BDAM method, an extension of BDAM and P-BDAM algorithms, was presented by Gobbetti et al in [GMC+06]. The method exploits a wavelet-based two stage near-lossless compression technique to efficiently encode the height map data. In C-BDAM, uniform batch triangulations are used which do not adapt to local surface features. Regular triangulations typically generate significantly more triangles and unreasonably increase the GPU load.

Terrain rendering method presented by Schneider and Westermann [SW06] partitions the terrain into square tiles and builds for each tile a discrete set of LODs using a nested mesh hierarchy. Following this approach, Dick et al proposed the method for tile triangulations encoding that enables efficient GPU-based decoding [DSW09].

All these methods either completely ignore local terrain surface features (like [LC03, LH04, GMC+06]) for the sake of efficient GPU utilization, or pre-compute the triangulations off-line and then just load them during rendering [CGG+03a, CGG+03b]. For the case of compressed data, GPU can also be used for geometry decompressing as well [SW06, DSW09].

By the best of our knowledge, none of the previous methods take an advantage of the tessellation unit exposed by the latest DX11-class graphics hardware for precise yet adaptive (view-dependent) terrain tessellation.

3. CONTRIBUTION

The main contribution is a novel terrain rendering approach, which combines efficiency of the chunk-based terrain rendering with the accuracy of fine-grain triangulation construction methods. In contrast to the previous approaches, our adaptive view-dependent triangulation is constructed entirely on the GPU using hardware-supported tessellation. This offloads computations from the CPU while also reduces expensive CPU-GPU data transfers. We also propose fast and simple GPU-accelerated compression technique for progressively encoding multiresolution hierarchy that enables direct control of a reconstruction precision.

Algorithm Overview

To achieve real-time rendering and meet the hardware limitations, we exploit the LOD technique. To create various levels of detail, during the preprocessing, a multiresolution hierarchy is constructed by recursively downsampling the initial data and subdividing it into overlapping patches. In order to reduce the memory requirements, the resulting hierarchy is then encoded using simple and efficient compression algorithm described in section 4.

Constructing adaptive terrain model to be rendered is a two-stage process. The first stage is the coarse per-patch LOD selection: the rendering algorithm selects the coarsest level patches that tolerate the given screen-space error. They are cached in a GPU memory and due to the frame-to-frame coherence are re-used for a number of successive frames. On the second stage, a fine-grain LOD selection is performed: each patch is seamlessly triangulated using hardware. For this purpose, each patch is subdivided into the equal-sized smaller blocks that are independently triangulated by the GPU-supported tessellation unit, as described in section 5.

Experimental results are given in section 6. Section 7 concludes the paper.

4. BUILDING COMPRESSED MULTIRESOLUTION TERRAIN REPRESENTATION

Patch Quad Tree

The core structure of the proposed multiresolution model is a quad tree of square blocks (hereinafter referred to as patches). This structure is commonly used in real-time terrain rendering systems [Ulr00, DSW09].

The patch quad tree is constructed at the preprocess stage. At the first step, a series of coarser height maps is built. Each height map is the downsampled version of the previous one (fig. 1). At the next step, the patch quad tree itself is constructed by subdividing each level into $(2^n+1)\times(2^n+1)$ square blocks (65x65, 129x129, 257x257 etc.), refer to fig. 2.

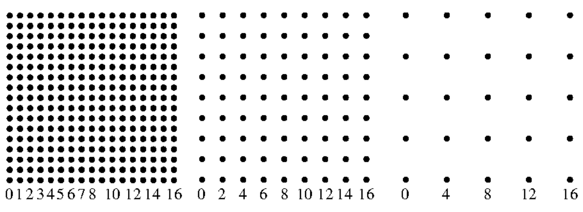


Figure 1. Downsampling initial height map.

Each patch in the quad tree hierarchy approximates the same area as its four children but with lower accuracy. To eliminate cracks, each patch shares one-sample boundary with its neighbors (hence $2^n + 1$ size).

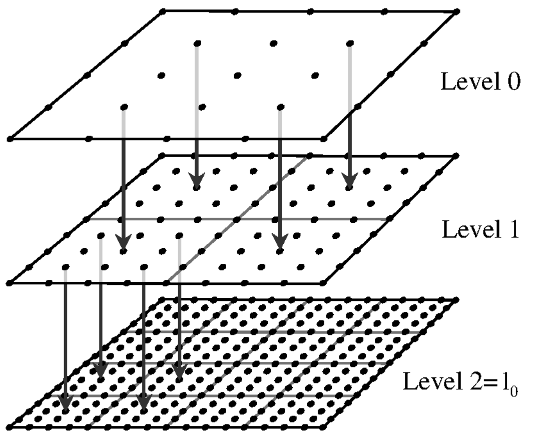


Figure 2. Patch quad tree.

The hierarchy is progressively encoded in a top-down order such that each patch's reconstruction error in L^∞ metric is bounded by the given error tolerance.

Quantizing Height Maps

Let's denote a sample in the l-th level located at the (i, j) position by $h_{i,j}^{(l)}$. Note that since level l-1 is simply the downsampled version of the level l, the following relation is always true: $h_{i,j}^{(l-1)} \equiv h_{2i,2j}^{(l)}$.

During the compression process, each level l of the hierarchy is quantized using a uniform quantizer with a dead zone (see fig. 3) as follows:

$$\hat{h}_{i,j}^{(l)} = \lfloor (h_{i,j}^{(l)} + \delta_l)/(2\delta_l) \rfloor \cdot 2\delta_l$$

where $\lfloor x \rfloor$ is rounding to the largest integer that is less than or equal to x, $\hat{h}_{i,j}^{(l)}$ is the quantized value, $\delta_l = \delta_{l_0} 2^{(l_0 - l)}$ is the maximum reconstruction error for the level l; l_0 is the finest resolution level number and $\delta_{l_0} > 0$ is the user-defined error tolerance for the finest level. Since our compression scheme is lossy, we assume that $\delta_{l_0} > 0$.

Quantized (integer) values $q_{i,j}^{(l)} = Q_l(h_{i,j}^{(l)})$ where $Q_l(h) = \lfloor (h + \delta_l)/(2\delta_l) \rfloor$ are lossless encoded as described below. The decoder reconstructs values as:

$$\hat{h}_{i,j}^{(l)} = 2\delta_l q_{i,j}^{(l)}$$

This quantization rule assures that for the l-th level, the maximum error is bounded by the δ_l:

$$\max_{i,j} | \hat{h}_{i,j}^{(l)} - h_{i,j}^{(l)} | \le \delta_l$$

The quantized values $\{q_{i,j}^{(0)}\}$ of the coarsest patch (located at the level 0) are encoded using adaptive arithmetic coding [WNC87]. The remaining patches are then progressively encoded as described in the following subsection.

Progressively Encoding Quantized Height Maps

Let us consider a patch's quantized height map $\hat{H}_P^{(l-1)}$ at the level l-1, and its 4 children joined height map $\hat{H}_C^{(l)}$ at the level l. Note that the first height map is $(2^n + 1) \times (2^n + 1)$ in size, while the second one is $(2 \cdot 2^n + 1) \times (2 \cdot 2^n + 1)$, both covering the same area. As it can be seen from fig. 1 and 2 (see also fig. 4), $\hat{H}_C^{(l)}$ shares the samples located at the even positions ((0,0), (0,2), (2,0) and so on) with $\hat{H}_P^{(l-1)}$. That is, the reconstructed sample $\hat{h}_{i,j}^{(l-1)}$ from the parent patch's height map $\hat{H}_P^{(l-1)}$ corresponds to the sample $\hat{h}_{2i,2j}^{(l)}$ in the $\hat{H}_C^{(l)}$. However $\hat{h}_{i,j}^{(l-1)}$ approximates the original (exact) value with the 2x lower accuracy than $\hat{h}_{2i,2j}^{(l)}$ should approximate and thus needs to be refined:

$$| \hat{h}_{i,j}^{(l-1)} - h_{i,j}^{(l-1)} | \le \delta_{l-1} = 2\delta_l$$

$$| \hat{h}_{2i,2j}^{(l)} - h_{2i,2j}^{(l)} | \le \delta_l$$

Recall that $h_{i,j}^{(l-1)} \equiv h_{2i,2j}^{(l)}$.

Our compression scheme consists of two steps. At the first step, we refine common samples of $\hat{H}_C^{(l)}$ and $\hat{H}_P^{(l-1)}$ (filled circles in fig. 4) to the required accuracy δ_l. At the second step, we encode the remaining samples (dotted circles in fig. 4) by interpolating the refined samples and encoding the prediction errors. Let's denote R to be the set of refined samples positions from $\hat{H}_C^{(l)}$ and I to be the set of interpolated samples positions:

$$R = \{ (i,j) : \hat{h}_{i,j}^{(l)} \in \hat{H}_C^{(l)} \wedge i = 2s, j = 2t, \ s,t \in Z \}$$

$I = \{(i,j) : \hat{h}_{i,j}^{(l)} \in \hat{H}_C^{(l)} \wedge (i,j) \notin R\}$

To refine samples from R, we exploit the following observation: the refined sample $\hat{h}_{2i,2j}^{(l)}$ (from $\hat{H}_C^{(l)}$) corresponding to the sample $\hat{h}_{i,j}^{(l-1)}$ (from $\hat{H}_P^{(l-1)}$) can only take one of the following 3 values (see fig. 3):

$\hat{h}_{2i,2j}^{(l)} \in \{\hat{h}_{i,j}^{(l-1)} - 2\delta_l, \hat{h}_{i,j}^{(l-1)}, \hat{h}_{i,j}^{(l-1)} + 2\delta_l\}$.

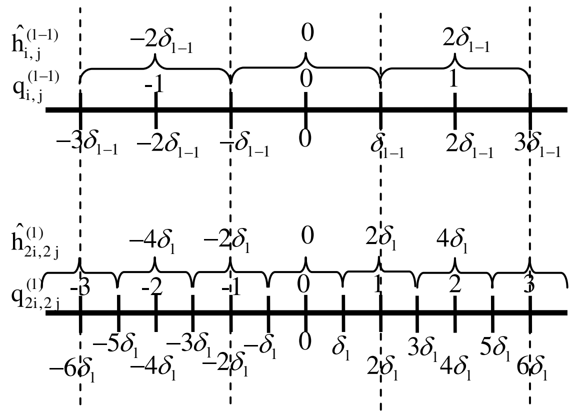


Figure 3. Quantizing two successive levels.

This also means that if $\hat{h}_{i,j}^{(l-1)}$ is encoded by the quantized value $q_{i,j}^{(l-1)}$, then corresponding $q_{2i,2j}^{(l)}$ can only take one of the following 3 values:

$q_{2i,2j}^{(l)} \in \{2q_{i,j}^{(l-1)} - 1,\ 2q_{i,j}^{(l-1)},\ 2q_{i,j}^{(l-1)} + 1\}$

Since $q_{i,j}^{(l-1)}$ is known, encoding the $q_{2i,2j}^{(l)}$ requires only 3 symbols: -1, 0 or $+1$. These symbols are encoded using adaptive arithmetic coding [WNC87].

At the second step, we encode the remaining samples located at positions from I in $\hat{H}_C^{(l)}$ (dotted circles in fig. 4). This is done by predicting the sample's value from the refined samples and by encoding the prediction error.

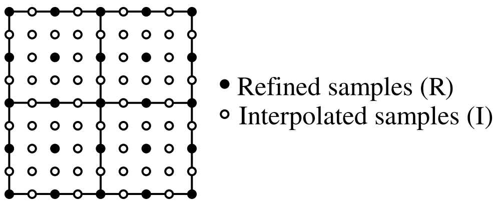


Figure 4. Refined and interpolated samples of the child patches joined height map $\hat{H}_C^{(l)}$.

For the sake of GPU-acceleration, we exploit bilinear predictor $P_R(\hat{h}_{i,j}^{(l)})$ that calculates predicted value of $\hat{h}_{i,j}^{(l)}$ as a weighted sum of 4 refined samples located at the neighboring positions in R. We then calculate the prediction error as follows:

$d_{i,j}^{(l)} = Q_l(P_R(\hat{h}_{i,j}^{(l)})) - q_{i,j}^{(l)}$, $(i,j) \in I$

Magnitudes and signs of the resulting prediction errors $d_{i,j}^{(l)}$ are then separately encoded using adaptive arithmetic coding.

As it was already discussed, symbols being used during described compression process are encoded with the technique described in [WNC87]. We exploit adaptive approach that learns the statistical properties of the input symbol stream on the fly. This is implemented as a histogram which counts corresponding symbol frequencies (see [WNC87] for details). Note that simple context modeling can improve the compression performance with minimal algorithmic complexity increase.

During the preprocessing, the whole hierarchy is recursively traversed starting from the root (level 0) and the proposed encoding process is repeated for each patch.

The proposed compression scheme enables direct control of the reconstruction precision in L^∞ error metric: it assures that the maximum reconstruction error of a terrain block at level l of the hierarchy is no more than δ_l. For comparison, compression method [Mal00] used in geometry clipmaps [LH04] does not provide a guaranteed error bound in L^∞ metric. C-BDAM [GMC+06] exploits sophisticated two-stage compression scheme to assure the given error tolerance. This provides higher compression ratios but is more computationally expensive than the presented scheme. Moreover, as we will show in the next section, our technique can be efficiently accelerated using the GPU.

Calculating Guaranteed Patch Error Bound

During the quad tree construction, each patch in the hierarchy is assigned a world space approximation error. It conservatively estimates the maximum geometric deviation of the patch's reconstructed height map from the underlying original full-detail height map. This value is required at the run time to estimate the screen-space error and to construct the patch-based adaptive model, which approximates the terrain with the specified screen-space error.

Let's denote the patch located at the level l of the quad tree at the (m, n) position by the $P_{m,n}^{(l)}$ and its upper error bound by the $Err(P_{m,n}^{(l)})$. To calculate $Err(P_{m,n}^{(l)})$, we first calculate approximation error $Err_{Appr}(P_{m,n}^{(l)})$, which is the upper bound of the maximum distance from the patch's <u>precise</u> height map to the samples of the underlying full-detail (level

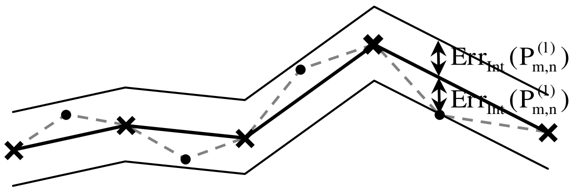
l_0) height map. It is recursively calculated using the same method as used in ROAM [DWS+97] to calculate the nested wedgie thickness:

$$\mathrm{Err}_{\mathrm{Appr}}(P^{(l_0)}_{m,n}) = 0$$

$$\mathrm{Err}_{\mathrm{Appr}}(P^{(l)}_{m,n}) = \mathrm{Err}_{\mathrm{Int}}(P^{(l)}_{m,n}) + \max_{s,t=\pm1}\{\mathrm{Err}_{\mathrm{Appr}}(P^{(l+1)}_{2m+s,2n+t})\},$$

$$l = l_0 - 1, ..0$$

where $\mathrm{Err}_{\mathrm{Int}}(P^{(l)}_{m,n})$ is the maximum geometric deviation of the linearly interpolated patch's height map from its children height maps. Two-dimensional illustration for determining $\mathrm{Err}_{\mathrm{Int}}(P^{(l)}_{m,n})$ is given in fig. 5.


$\mathrm{Err}_{\mathrm{Int}}(P^{(l)}_{m,n})$
$\mathrm{Err}_{\mathrm{Int}}(P^{(l)}_{m,n})$

● Child patches' (level l) height map samples

✕ Parent patch's (level l-1) height map samples

Figure 5. Patch's height map interpolation error.

Since for the patch $P^{(l)}_{m,n}$, the reconstructed height map deviates from the exact height map by at most δ_l, the final patch's upper error bound is given by:

$$\mathrm{Err}(P^{(l)}_{m,n}) = \mathrm{Err}_{\mathrm{Appr}}(P^{(l)}_{m,n}) + \delta_l$$

5. CONSTRUCTING VIEW-DEPENDENT ADAPTIVE MODEL

The proposed level-of-detail selection process consists of two stages. The first stage is the coarse LOD selection which is done on a per-patch level: an unbalanced patch quad tree is constructed with the leaf patches satisfying the given error tolerance. On the second stage, the fine-grain LOD selection is performed, at which each patch is precisely triangulated using the hardware tessellation unit.

Coarse Level of Detail Selection

The coarse LOD selection is performed similar to other quad tree-based terrain rendering methods. For this purpose, an unbalanced patch quad tree is maintained. It defines the block-based adaptive model, which approximates the terrain with the specified screen-space error.

The unbalanced quad tree is cached in a GPU memory and is updated according to the results of comparing patch's screen-space error estimation $\mathrm{Err}_{\mathrm{Scr}}(P^{(l)}_{m,n})$ with the user-defined error threshold ε. Since we already have the maximum geometric error for the vertices within a patch, $\mathrm{Err}_{\mathrm{Scr}}(P^{(l)}_{m,n})$ can be calculated using standard LOD formula for conservatively determining the maximum screen-space vertex error (see [Ulr00, Lev02]):

$$\mathrm{Err}_{\mathrm{Scr}}(P^{(l)}_{m,n}) = \gamma \frac{\mathrm{Err}(P^{(l)}_{m,n})}{\rho(c, V^{(l)}_{m,n})}$$

where $\gamma = \frac{1}{2}\max(R_h \cdot \mathrm{ctg}(\varphi_h/2),\ R_v \cdot \mathrm{ctg}(\varphi_v/2))$, R_h and R_v are horizontal and vertical resolutions of the view port, φ_h and φ_v are the horizontal and vertical camera fields of view, and $\rho(c, V^{(l)}_{m,n})$ is the distance from the camera position c to the closest point on the patch's bounding box $V^{(l)}_{m,n}$.

Tessellation Blocks

During the fine-grain LOD selection, each patch in the unbalanced patch quad tree is adaptively triangulated using the GPU. For this purpose, each patch is subdivided into the small equal-sized blocks that we call tessellation blocks. For instance, a 65×65 patch can be subdivided into the 4×4 grid of 17×17 tessellation blocks or into the 8×8 grid of 9×9 blocks etc. Detail level for each tessellation block is determined independently by the hull shader: the block can be rendered in the full resolution (fig. 6, left) or in the resolution reduced by a factor of 2^d, d = 1,2,… (fig. 6, center, right).

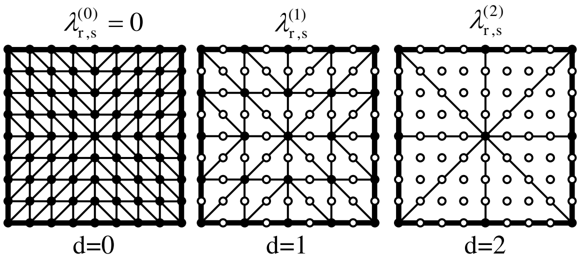


Figure 6. Triangulations of a 9×9 tessellation block.

To determine the degree of simplification for each block, we calculate a series of block errors. These errors represent the deviation of the block's simplified triangulation from the patch's height map samples, covered by the block but not included into the simplified triangulation (dotted circles in fig. 6).

Let's denote the error of the tessellation block located at the (r, s) position in the patch, whose triangulation is simplified by a factor of 2^d by $\lambda^{(d)}_{r,s}$. The tessellation block errors $\lambda^{(d)}_{r,s}$ are computed as follows:

$$\lambda^{(d)}_{r,s} = \max_{v \notin T^{(d)}_{r,s}} \rho(v, T^{(d)}_{r,s}),\ d = 1,2,...$$

where $T^{(d)}_{r,s}$ is the tessellation block (r,s) triangulation simplified by a factor of 2^d and $\rho(v,T^{(d)}_{r,s})$ is the vertical distance from the vertex v to the triangulation $T^{(d)}_{r,s}$. Two and four times simplified triangulations as well as these samples (dotted circles) of the patch's height map that are used to calculate $\lambda^{(1)}_{r,s}$ and $\lambda^{(2)}_{r,s}$ are shown in fig. 6 (center and right images correspondingly).

To get the final error bound for the tessellation block, it is necessary to take into account the patch's error bound. This final error bound hereinafter is referred to as $\Lambda^{(d)}_{r,s}$ and is calculated as follows:

$$\Lambda^{(d)}_{r,s} = \lambda^{(d)}_{r,s} + \mathrm{Err}(P^{(l)}_{m,n})$$

In our particular implementation, we calculate errors for 4 simplification levels such that tessellation block triangulation can be simplified by a maximum factor of $(2^4)^2 = 256$. This enables us to store the tessellation block errors as a 4-component vector.

Fine-Grain Level of Detail Selection

When the patch is to be rendered, it's necessary to estimate how much its tessellation blocks' triangulations can be simplified without introducing unacceptable error. This is done using the current frame's world-view-projection matrix. Each tessellation block is processed independently and for each block's edge, a tessellation factor is determined. To eliminate cracks, tessellation factors for shared edges of neighboring blocks must be computed in the same way. The tessellation factors are then passed to the tessellation stage of the graphics pipeline, which generates final triangulation.

Tessellation factors for all edges are determined identically. Let's consider some edge and denote its center by e_c. Let's define edge errors $\Lambda^{(d)}_{e_c}$ as the maximum error of the tessellation blocks sharing this edge. For example, block (r, s) left edge's errors are calculated as follows:

$$\Lambda^{(d)}_{e_c} = \max(\Lambda^{(d)}_{r-1,s}, \Lambda^{(d)}_{r,s}),\ d = 1,2,\ldots$$

Next let's define a series of segments in a world space specified by theirs end points $e^{(d),-}_c$ and $e^{(d),+}_c$ determined as follows:

$$e^{(d),-}_c = e_c - \Lambda^{(d)}_{e_c} / 2 \cdot e_z$$

$$e^{(d),+}_c = e_c + \Lambda^{(d)}_{e_c} / 2 \cdot e_z$$

where e_z is the world space z (up) axis unit vector.

Thus $e^{(d),-}_c$ and $e^{(d),+}_c$ define a segment of length $\Lambda^{(d)}_{e_c}$ directed along the z axis such that the edge centre e_c is located in the segment's middle.

If we project this segment onto the viewing plane using the world-view-projection matrix, we will get the edge screen space error estimation (fig. 7) given that the neighboring tessellation blocks are simplified by a factor of 2^d. We can then select the maximum simplification level d for the edge that does not lead to unacceptable error as follows:

$$d = \arg\max_d \ \mathrm{proj}(\overline{e^{(d),-}_c, e^{(d),+}_c}) < \varepsilon$$

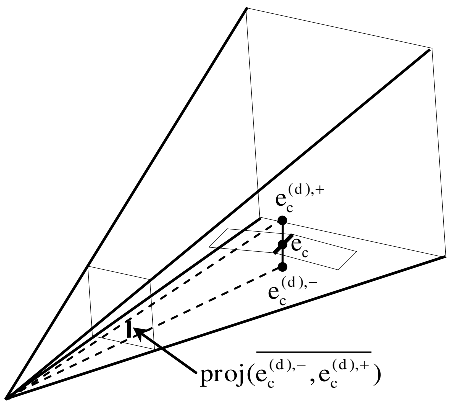


Figure 7. Calculating edge screen space error.

The same selection process is done for each edge. Tessellation factor for the block interior is then defined as the minimum of its edge tessellation factors. This method assures that tessellation factors for shared edges of neighboring blocks are computed equally and guarantees seamless patch triangulation. An example of a patch triangulation is given in fig. 8.

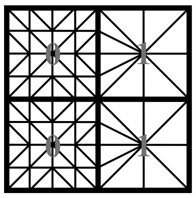


Figure 8. Seamlessly triangulated patch's tessellation blocks.

To hide gaps between neighboring patches, we exploit "vertical skirts" around the perimeter of each patch as proposed by T. Ulrich [Ulr00]. The top of the skirt matches the patch's edge and the skirt height is selected such that it hides all possible cracks.

Note that in contrast to all previous terrain simplification methods, all operations required to triangulate the patch are performed entirely on the GPU and does not involve any CPU computations.

Implementation Details

The presented algorithm was implemented with the C++ in an MS Visual Studio .NET environment.

In our system, the CPU decodes the bit stream in parallel to the rendering thread and all other tasks are done on the GPU. To facilitate GPU-accelerated decompression, we support several temporary textures. The first one is $(2^n+1)\times(2^n+1)$ 8-bit texture T_R that is populated with the parent patch's refinement labels (-1, 0 or $+1$) from R. The second one is $(2\cdot 2^n+1)\times(2\cdot 2^n+1)$ 8-bit texture T_I storing prediction errors $d^{(1)}_{i,j}$ for samples from I. GPU-part of the decompression is done in two steps:

- At the first step, parent patch height map is refined by rendering to the temporary texture T_P.
- At the second step, child patch height maps are rendered.

During the second step, T_P is filtered using hardware-supported bilinear filtering, interpolation errors are loaded from T_I and added to the interpolated samples from T_P.

Patch's height and normal maps as well as the tessellation block errors are stored as texture arrays. A list of unused subresources is supported. When patch is created, we find unused subresource in the list and release it when the patch is destroyed. Tessellation block errors as well as normal maps are computed on the GPU when the patch is created by rendering to the appropriate texture array element.

Exploiting texture arrays enables the whole terrain rendering using single draw call with instancing. The per-instance data contains patch location, level in the hierarchy and the texture index. Patch rendering hull shader calculates tessellation factor for each edge and passes the data to the tessellator. Tessellator generates topology and domain coordinates that are passed to the domain shader. Domain shader calculates world space position for each vertex and fetches the height map value from the appropriate texture array element. The resulting triangles then pass in a conventional way via rasterizer.

6. EXPERIMENTAL RESULTS AND DISCUSSION

To test our system, we used 16385×16385 height map of the Puget Sound sampled at 10 meter spacing, which is used as the common benchmark and is available at [PS]. The raw data size (16 bps) is 512 MB. The compression and run-time experiments were done on a workstation with the following hardware configuration: single Intel Core i7 @2.67; 6.0 GB RAM; NVidia GTX480.

The data set was compressed to 46.8 MB (11:1) with 1 meter error tolerance. For comparison, C-BDAM method, which exploits much more sophisticated approach, compressed the same data set to 19.2 MB (26:1) [GMC+06]. Note that in contrast to C-BDAM, our method enables hardware-based decompression. Note also that in practice we compress extended $(2^n+3)\times(2^n+3)$ height map for each patch for the sake of seamless normal map generation. As opposed to compressing conventional diffuse textures, height maps usually require less space. That is why we believe that provided 11x compression rate is a good justification for the quality of our algorithm.

During our run-time experiments, the Puget Sound data set was rendered with 2 pixels screen space error tolerance at 1920x1200 resolution (fig. 10). We compared the rendering performance of our method with our implementation of the chunked LOD approach [Ulr00]. As fig. 10 shows, the data set was rendered at **607** fps on average with minimum at **465** fps with the proposed method. When the same terrain was rendered with our method but without exploiting instancing and texture arrays described previously, the frame rate was almost **2x** lower. As fig. 10 shows, our method is more than **3.5x** faster than the chunked LOD approach.

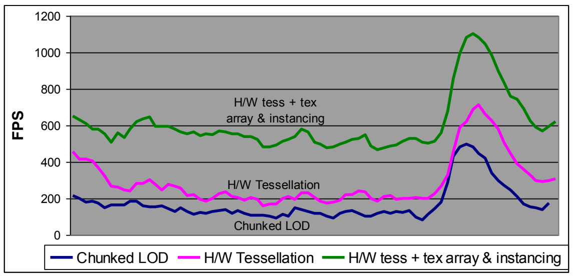


Figure 10. Rendering performance at 1920×1200 resolution.

Our experiments showed that the optimal tessellation block size that provides the best performance is 8×8. Other interesting statistics for this rendering experiment is that approximately 1024 of 128×128 patches were kept in GPU memory (only ~200 of them were rendered per frame on average). Each height map was stored with 16 bit precision. All patches demanded just 32 MB of the GPU memory. We also exploited normal map compressed using BC5, which required additional 16 MB of data. Diffuse maps are not taken into account because special algorithms that are behind the scope of this work are designed to compress them. However, the same quad tree-based subdivision scheme can be integrated with our method to handle diffuse texture.

Since our method enables using small screen space error threshold (2 pixels or less), we did not observe any popping artifacts during our experiments even though there is no morph between successive LODs in our current implementation.

In all our experiments, the whole compressed hierarchy easily fitted into the main memory. However, our approach can be easily extended for the out-of-core rendering of arbitrary large terrains. In this case, the whole compressed multiresolution representation would be kept in a repository on the disk or a network server, as for example in the geometry clipmaps. This would allow on-demand extraction from the repository rather accessing the data directly in the memory.

7. CONCLUSION AND FUTURE WORK

We presented a new real-time large-scale terrain rendering technique, which is based on the exploitation of the hardware-supported tessellation available in modern GPUs. Since triangulation is performed entirely on the GPU, there is no need to encode the triangulation topology. Moreover, the triangulation is precisely adapted to each camera position. To reduce the data storage requirements, we use robust compression technique that enables direct control over the reconstruction precision and is also accelerated by the GPU.

We consider support for dynamic terrain modifications as a future work topic. Since the triangulation topology is constructed entirely on the GPU, it would require only updating the tessellation block errors, and the triangulation will be updated accordingly. Another possible direction is to extend the presented algorithm for rendering arbitrary high-detailed 2D-parameterized surfaces.

8. REFERENCES

[CGG+03a] Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., and Scopigno, R. BDAM – batched dynamic adaptive meshes for high performance terrain visualization. Computer Graphics Forum, Vol. 22, No. 3, pp. 505–514, 2003.

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

[DSW09] Dick, C., Schneider, J., and Westermann, R. Efficient Geometry Compression for GPU-based Decoding in Realtime Terrain Rendering. In Computer Graphics Forum, Vol. 28, No 1, pp. 67–83, 2009.

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

[GMC+06] Gobbetti, E., Marton, F., Cignoni, P., Di Benedetto, M., and Ganovelli, F. C-BDAM – compressed batched dynamic adaptive meshes for terrain rendering. Computer Graphics Forum, Vol. 25, No. 3, pp. 333–342, 2006.

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

[LC03] Larsen, B.D., and Christensen, N.J. Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail. Journal of WSCG, Vol. 11, No. 2, pp. 282–289, 2003.

[Lev02] Levenberg, J. Fast view-dependent level-of-detail rendering using cached geometry. In Proc. IEEE Visualization, pp. 259–265, 2002.

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

[LH04] Losasso, F., and Hoppe, H. Geometry clipmaps: Terrain rendering using nested regular grids. In Proc. ACM SIGGRAPH, pp. 769–776, 2004.

[Mal00] Malvar, H. Fast Progressive Image Coding without Wavelets. In Proceedings of Data Compression Conference (DCC '00), Snowbird, UT, USA, pp. 243–252, 28-30 March 2000.

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

[PG07] Pajarola, R., and Gobbetti, E. Survey on semi-regular multiresolution models for interactive terrain rendering. The Visual Computer, Vol. 23, No. 8, pp. 583–605, 2007.

[PS] Puget Sound elevation data set is available at http://www.cc.gatech.edu/projects/large_models/ps.html

[SW06] Schneider, J., and Westermann, R. GPU-Friendly High-Quality Terrain Rendering. Journal of WSCG, Vol. 14, pp. 49–56, 2006.

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

[WNC87] Witten, I.H., Neal, R.M., and Cleary J.G., Arithmetic coding for data compression. Comm. ACM, Vol. 30, No. 6, pp. 520–540, June 1987.