Volume xx (200y), Number z, pp. 1–16

Efficient Geometry Compression for GPU-Based Decoding in Realtime Terrain Rendering

Christian Dick, Jens Schneider, and Rüdiger Westermann

Computer Graphics & Visualization Group
Technische Universität München

Abstract

We present a geometry compression scheme for restricted quadtree meshes and use this scheme for the compression of adaptively triangulated digital elevation models (DEMs). A compression factor of 8-9 is achieved by employing a generalized strip representation of quadtree meshes to incrementally encode vertex positions. In combination with adaptive error-controlled triangulation this allows us to significantly reduce bandwidth requirements in the rendering of large DEMs that have to be paged from disk. The compression scheme is specifically tailored for GPU-based decoding since it minimizes dependent memory access operations. We can thus trade CPU operations and CPU-GPU data transfer for GPU processing, resulting in twice faster streaming of DEMs from main memory into GPU memory. A novel storage format for decoded DEMs on the GPU facilitates a sustained rendering throughput of about 300 million triangles per second. Due to these properties, the proposed scheme enables scalable rendering with respect to the display resolution independent of the data size. For a maximum screen-space error below one pixel it achieves frame rates of over 100 fps, even on high resolution displays. We validate the efficiency of the proposed method by presenting experimental results on scanned elevation models of several hundred gigabytes.

Categories and Subject Descriptors (according to ACM CCS): E.4 [Data]: Coding and Information Theory - Data Compression I.3.3 [Computer Graphics]: Picture/Image Generation - Viewing Algorithms I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - Surface Representations, Geometric Algorithms, Object Hierarchies I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism - Virtual Reality

1. Introduction

Advances in the science and technology of remote sensing have led to a sheer explosion of the size and resolution of digital elevation models (DEMs). Today, spatial resolutions as high as 1 meter are available for some areas, and data at only slightly lower resolution is currently being acquired for ever larger regions worldwide. Figure 1 shows a DEM of the State of Utah, which covers a 460 km $\times$ 600 km area at a resolution of 5 meters. This already amounts to over 20 GB of raw data, excluding additional photo textures that are typically acquired at an even higher resolution. From this observation it becomes clear that the challenge in terrain rendering today is to maintain interactive frame rates for data sets that vastly exceed the available main memory. In particular on high-resolution displays, where the number of primitives to be sent to the GPU for rendering is significantly increasing, too, terrain rendering performance is limited by bandwidth restrictions rather than computational or rendering power.

Meeting these requirements on recent PC architectures is still challenging because of limitations in memory and communication bandwidth. As the data is too large to fit into main memory, the rendering process involves frequent disk access. Today it is well accepted that this operation is the most important aspect of terrain rendering performance, and considerable effort has been spent on reducing the amount of data to be loaded from disk. The method that has been doing this most rigorously is the geometric clipmap approach proposed by Losasso and Hoppe [LH04]. As an alternative to error controlled re-meshing of the terrain height field, a regular height map pyramid is used to compress large DEMs by as much as a factor of one hundred. Geometric clipmaps are very effective in reducing expensive disk access, but on

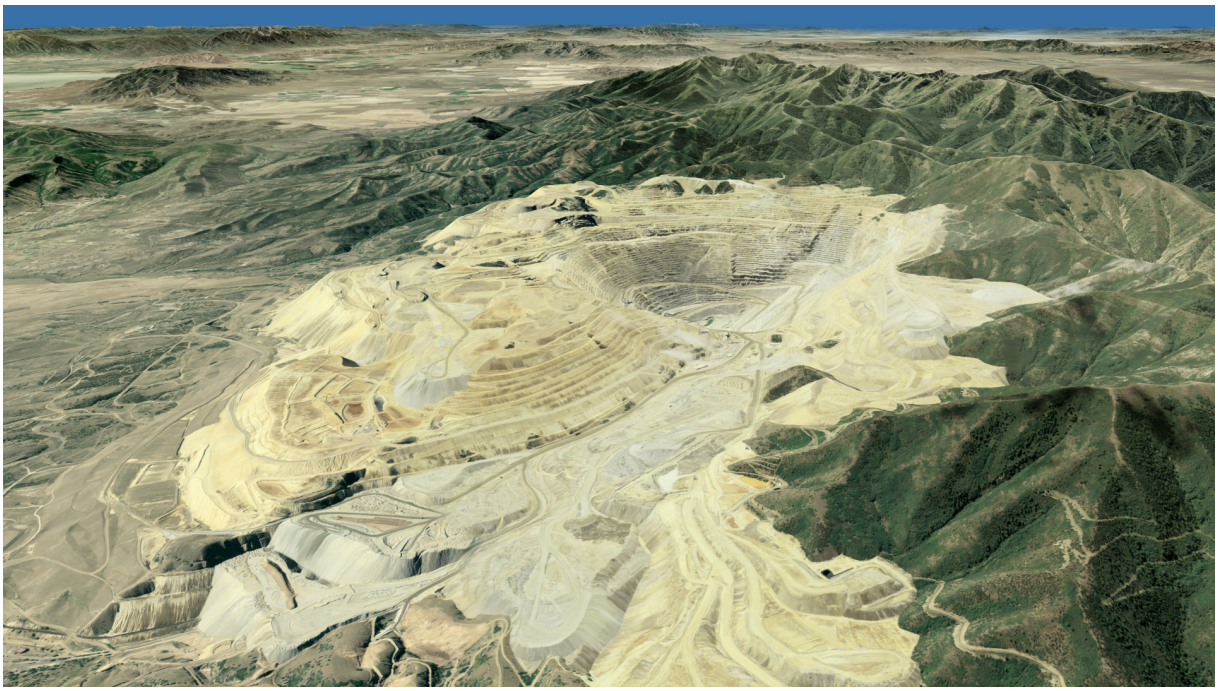


Figure 1: *A textured DEM of the State of Utah (460 km × 600 km) is rendered on a 1920 × 1080 view port using our method. The spatial resolution of the DEM and the texture is 5 m and 1 m, respectively. Even though we render to a 2 megapixel view port, an average frame rate of over 135 fps is achieved at a geometric screen-space error of below one pixel.*

the other hand they require uniform mesh tesselation independent of the height field variation. Therefore, in particular at high display resolution and low pixel-error bound, they tend to increase the number of height values that have to be uploaded to the GPU as well as the number of rendered triangles.

2. Contribution

The primary focus of this paper is the development of a compression scheme for large DEMs to reduce disk access and CPU-GPU data transfer in terrain rendering. The method we propose works on restricted quadtree meshes, which are generated by error-controlled adaptive tesselation of an elevation model. This allows us to significantly reduce the number of triangles required to represent the height field and to precisely specify the maximum deviation of the rendered field from the initial elevation model.

The compression scheme we present reduces the memory requirements of restricted quadtree meshes by a factor of 8-9, and it thus allows us to reduce bandwidth requirements when large DEMs have to be streamed from disk. The scheme builds on a generalized strip representation of such quadtree meshes to incrementally encode vertex positions. It is specifically tailored for GPU-based decoding since it avoids the explicit depth-first traversal of the quadtree hierarchy and allows for immediate decoding of elements in the compressed stream. In particular, it does not require any associated index structure and can thus avoid dependent memory access operations on the GPU.

Due to this particular design, our method can make exhaustive use of the GPU and trades disk access, CPU operations, and CPU-GPU data transfer for GPU processing. By effectively exploiting computation and bandwidth capacities on recent GPUs, bandwidth limitations in the rendering of large DEMs can be greatly alleviated. Even though data decoding is performed on the GPU, our method maintains a sustained throughput of about 300 million triangles per second. This amounts to a frame rate of over 100 frames per second, independently of the extent and resolution of the data set to be rendered. This throughput is achieved even at high image resolutions of several megapixels *and* a maximum screen-space error below one pixel. The approach runs on low-end single-core PCs and does not require any sophisticated disk technology.

We have integrated the proposed compression scheme into a tile-based visually continuous terrain rendering method, which renders very large DEMs and photo textures at high resolution (see Figure 1). To reduce the memory used by terrain photo textures, we employ the S3TC fixed-rate compression scheme [S3]. It has been established as the de facto standard for texture compression, with the DXT1 codec at 6:1 compression ratio being used in terrain rendering. Even though the S3TC scheme does not per-

form favorable to other compression schemes like JPEG and JPEG2000, both with respect to compression ratio and reconstruction quality, S3TC allows compressed textures to be stored and randomly accessed in GPU memory and supports anisotropic texture filtering.

The remainder of this paper is organized as follows: In the next section we outline previously published LOD techniques for terrain rendering. Next, we describe our novel geometry compression scheme for quadtree meshes, and we show how to make use of the GPU for mesh decoding. We then describe the memory management strategies underlying our terrain rendering method. In Section 8 we present a detailed analysis of all parts of this method. The paper is concluded with a discussion and remarks on future work.

3. Related Work

Over the last decade, a number of view-dependent LOD techniques for terrain rendering have been proposed, which differ mainly in the hierarchical structures used. Previous work can be classified into dynamic re-meshing strategies, region-based multi-resolution approaches, and regular nested grids, all of which allow for visually continuous LOD rendering. For a thorough overview of the field let us refer here to the recent survey by Pajarola and Gobbetti [PG07].

View-dependent refinement techniques construct a continuous LOD triangulation in every frame with respect to a given world-space deviation and screen-space error tolerance. Early approaches were based on triangulated irregular networks (TINs) as introduced by Peucker [PFL78] and Fowler [FL79] because of their approximation quality (see, for instance, the terrain rendering methods proposed by Garland and Heckbert [GH95] and Hoppe [Hop98]). Irregular triangulations minimize the amount of triangles to be rendered at a given approximation error, but on the other hand they require quite elaborate data structures making them rather CPU intense. Consequently, more regular triangulations have been used, for instance, bintree hierarchies [LKR*96, DWS*97] and restricted quadtree meshes [VB87, Paj98], both of which are based on longest edge bisection. Compared to TINs such triangulations tend to increase the number of polygons required to represent a DEM at a particular approximation quality, but they simplify the re-meshing process and can thus reduce CPU processing at run-time. By employing advanced traversal and data access strategies, (semi-)regular triangulations can be rebuilt from scratch in every frame even for large DEMs [LP01, LP02].

Region-based multi-resolution approaches partition the terrain into tiles that can be processed independently [KLR*95, SN95, Blo00]. To avoid any re-meshing at runtime, mesh hierarchies at varying approximation error are pre-computed and appropriate approximations are selected for rendering [RLIB99, Pom00, CGG*03a, HDJ04, SW06]. To avoid visual artifacts like popping, either geomorphs are used [FEKR90] or the maximum screen-space error is restricted to one pixel [CGG*03b]. To avoid inconsistencies at tile boundaries, additional constraints can either be incorporated into the error metric employed [RHSS98] or can directly be enforced at boundary edges [Pom00]; or boundaries can be patched using *skirts*, *flanges*, or *zero-area triangles* [Ulr02, WMD*04]. Partitioning the data into larger chunks that can be cached in GPU memory to reduce CPU-GPU bandwidth requirements as well as the number of draw calls on the GPU has been investigated in [Lev02, WMD*04, SW06].

Losasso and Hoppe [LH04] even show that re-meshing can entirely be avoided by using a set of nested regular grids centered about the viewer. As the grid resolution decreases with increasing distance to the viewer, approximately uniform screen-space resolution is achieved.

As scanned DEMs are typically much larger than the main memory available on desktop computers, data layout schemes on external storage devices [LP02, CGG*03b, CGG*03a] in combination with visibility-based and speculative pre-fetching strategies as well as GPU occlusion queries have been developed [CKS03]. Common to all these approaches is the goal to reduce both the amount of data to be transferred from disk and the number of seek operations to be performed. Ng et al. [NNT*05] provided quantitative evidence that a circle-shaped pre-fetching region is superior to a fan-shaped region, as it only requires slightly more memory but does not considerably increase bandwidth requirements, yet it allows fast changes of the viewing direction.

Along a different avenue of research, compression schemes for DEMs have been proposed, for instance, based on wavelet decomposition [GGS95, GMC*06], space-filling curves [Ger03], or tiled quadtrees [PD04]. Due to the regular grid pyramid used in the geometric clipmap approach [LH04], this technique enables lossy compression of digital height maps using standard image compression schemes. Note that compression schemes based on space-filling curves are similar in style to compression schemes for general meshes [TG98, TR98], since they both exploit spatial coherences along the specific traversal of the domain. However, compression schemes for DEMs greatly benefit from the more regular structure, and can thus achieve lower bitrates for both topology and geometry. A promising approach was presented by Gobetti et al. [GMC*06], who employed a near-lossless wavelet-based compression scheme. Since this method requires regular connectivity inside each batch, it only needs to encode the geometry, but not the topology of each batch. On the other hand, regular triangulations typically generate more triangles, and the use of wavelets makes error control in the L_∞ norm rather complicated.

It should be noted that although the aforementioned approaches can achieve similar or even better compression

rates than our method, CPU-based decoding of the compressed data streams is inherent to all of them. As a consequence, their throughput is limited by the computational power and memory bandwidth of the CPU as well as the bandwidth of the graphics bus.

4. Geometry Compression

Our geometry compression scheme works on restricted quadtree meshes [VB87, Paj98]. Mesh construction and compression is performed in a pre-processing step: Firstly, we build a Gaussian pyramid of the entire DEM. Starting with the original height field (associated with level number 0 in the pyramid), the coarser levels (associated with ascending level numbers) are successively computed by averaging blocks of 2×2 samples. Thus, at each coarser level the number of samples is reduced by a factor of 2 in each dimension, at the same time increasing the grid spacing by the same factor. Secondly, starting from the finest level of the pyramid, each level is tiled into square regions of $M \times M$ samples with one sample overlap to adjacent tiles. Tiling is performed up to the level where the entire domain fits into a single tile. In our current implementation a tile size of $M = 513$ is used. The tiles are finally organized as a quadtree, i.e., each tile covers exactly four tiles in the next finer level.

At each level ℓ, every tile is meshed separately with respect to a world-space error tolerance ε_ℓ. To obtain an isotropic simplification of the height field, this tolerance is equal to the level's grid spacing δ_ℓ, meaning that for a tile at level ℓ a restricted quadtree mesh with a deviation (measured in the L_∞ norm) of up to $\varepsilon_\ell := \delta_\ell$ meters to the original terrain model is built. Thus, at decreasing tile resolutions ever increasing error tolerances are considered. In particular, because $\delta_{\ell+1} = 2\,\delta_\ell$, it holds that $\varepsilon_\ell = 2^\ell \varepsilon_0$.

To allow for a vertical tolerance that is smaller than δ_0 (note that the vertical error of a DEM is typically smaller than its spacing), we create additional levels by meshing level 0 with respect to world-space error tolerances of $\frac{\delta_0}{2}, \frac{\delta_0}{4}, \ldots$. For these levels, the tile size is reduced to $\frac{M+1}{2} \times \frac{M+1}{2}, \frac{M+3}{4} \times \frac{M+3}{4}, \ldots$ samples per tile to preserve the quadtree structure. This strategy generates a LOD hierarchy that represents the DEM with maximum accuracy, but, because it adds additional levels, it reduces the overall compression rate.

If a particular region of the terrain should be rendered at a world-space error tolerance ε, the LOD ℓ to be used is determined by

$$\varepsilon_\ell \leq \varepsilon < \varepsilon_{\ell+1}, \tag{1}$$

i.e., the coarsest LOD with a world-space error tolerance of at most ε is selected. Since $\varepsilon_\ell = 2^\ell \varepsilon_0$, it follows that $2^\ell \leq \frac{\varepsilon}{\varepsilon_0} < 2^{\ell+1}$, and thus

$$\ell = \left\lfloor \log_2 \left(\frac{\varepsilon}{\varepsilon_0} \right) \right\rfloor.$$

Before a tile is finally meshed, the tile's height values are quantized. Using a uniform quantization at k bits, the quantization error is at most $\varepsilon_Q = \frac{1}{2} \frac{h_{\max} - h_{\min}}{2^k}$, where $h_{\min}$ and $h_{\max}$ denote the tile's minimum and maximum height value. Thus, the number of bits k that is required to guarantee a quantization error less or equal to ε_Q is computed as

$$k = \max \left\{ \left\lceil \log_2 \left(\frac{h_{\max} - h_{\min}}{2\varepsilon_Q} \right) \right\rceil, 0 \right\}.$$

To not exceed the overall error tolerance ε_ℓ of the tile's level ℓ, the quantization error tolerance ε_Q has to be within the interval $]0, \varepsilon_\ell]$. In all of our experiments, choosing $\varepsilon_Q := 0.25\,\varepsilon_\ell$ gave very good results, both in terms of the number of bits used to represent the height values and the number of generated triangles.

Mesh construction then works as proposed in [DWS*97]. Starting with an initial mesh consisting of four right angle triangles sharing the vertex at the right angle (see Figure 3), the mesh is successively refined by splitting triangles along the edge formed by the vertex at the right angle and the midpoint of the hypotenuse. To avoid T-vertices, splitting a triangle requires to simultaneously split the triangle sharing the hypotenuse. This split has to be recursively repeated until a consistent triangulation is obtained, i.e., until the edge to be split is the hypotenuse of both triangles or a boundary edge.

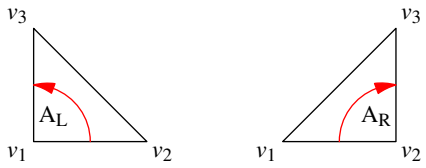
To build a tile's mesh, we successively split triangles with a deviation from the original height field that is larger than the error tolerance ε_ℓ, until the generated mesh is within the error tolerance. To guarantee that the total error does not exceed ε_ℓ, we compute the deviation of the mesh based on quantized height values from the original height field *before* quantization. During mesh construction, a generalized triangle strip representation is built for each mesh. This is outlined in the following section.

4.1. Mesh Serialization

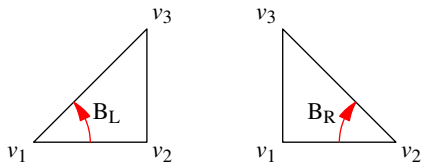
The compression scheme we propose for restricted quadtree meshes is similar in spirit to the method presented in [Ger03], but it avoids the depth-first traversal of the bintree hierarchy and is specifically tailored to meet the demand for GPU-based decoding. Our method builds a generalized triangle strip, and it thus requires to store only one new vertex per triangle. In the following we show that by using our compression scheme a triangle can be represented by the bits used to encode the height value of the new vertex and two additional bits to encode the *x*/*y*-coordinates of this vertex.

To generate a generalized strip representation of the triangle mesh, we construct a directed path that visits each triangle exactly once and enters/leaves triangles only across edges. As shown in Figure 2, triangles can then be classified into six different cases depending on the edges across which the path enters and leaves the triangle. The type (A, B, C) is determined by the type of the entering and the leaving edge

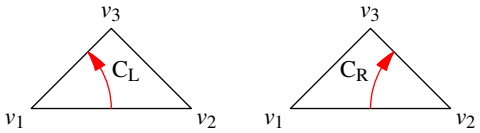
(cathetus or hypotenuse). The winding (L, R) specifies the path's direction as either left or right winding.



Type A: From cathetus to cathetus



Type B: From cathetus to hypotenuse



Type C: From hypotenuse to cathetus

Figure 2: *The six triangle cases are determined by the edges across which the path enters and leaves the triangle as well as the winding.*

The path construction we perform is a variation of the method proposed by Pajarola [Paj98]. The construction is directly incorporated into the construction of the restricted quadtree mesh. Starting with a mesh and a path as shown in Figure 3, the replacement system shown in Figure 4 is used to simultaneously build the path and the quadtree mesh. By using Follow sets from formal language theory, it can be shown easily that the replacement system is closed, i.e., the given rules are sufficient to construct every possible restricted quadtree mesh. On the other hand, not all replacement rules may have to be used.

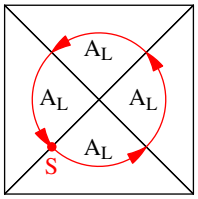


Figure 3: *The initial mesh that is successively subdivided to generate a restricted quadtree mesh.* S *indicates the start/end of the path that sequentially traverses all triangles of the mesh.*

The idea underlying our compression method is to build the triangle strip using only triangle types, windings, and height values. Given the two vertices of the entering edge of a triangle, the third vertex of the triangle is uniquely determined by the triangle type, the winding, and the height

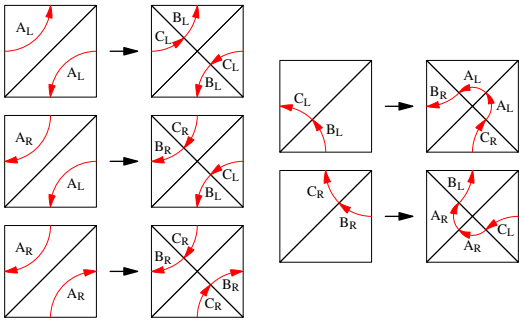


Figure 4: *The replacement system used to simultaneously build the restricted quadtree mesh and the directed path (red). Only the respective half of a replacement rule is applied if the diagonal lies on the border of the tile.*

value of this vertex. The winding also determines the leaving edge, so that the two vertices of the entering edge of the next triangle are known and the construction process can be repeated until all triangles have been visited.

The Follow sets also show that each triangle case can be followed only by one of the two possible windings for every triangle type, e.g., A_L can only be followed by A_L, B_R or C_L, but not by A_R, B_L and C_R (see the appendix for a full table). Since the winding can be determined from the type and winding of the preceding triangle and the type of the current triangle, it is thus sufficient to store the type of a triangle. The vertices of the entering edge and the winding of the very first triangle have to be stored explicitly.

Instead of using one strip to encode the entire mesh, we divide the strip into sub-strips of length n. Therefore, for every $n^{\text{th}}+1$ triangle along the path (i.e., for the 1^{st}, $(n+1)^{\text{st}}$, $(2n+1)^{\text{st}}$,... triangle) the vertices of its entering edge and its winding are stored. In this way, the mesh can be decoded in parallel, for instance on recent GPUs, by simultaneously starting with the first and every $n^{\text{th}}+1$ triangle. In the current implementation, a sub-strip length of $n = 16$ is used.

4.2. Data Layout

The compressed geometry of each tile is stored in two parts. The first part is a list of entries, one per sub-strip, each encoding the vertices of the entering edge and the winding of the first triangle of the sub-strip. The second part is the list of triangles traversed along the path, each specified by its triangle type and the height value of the new vertex. We refer to the first and second part as the *"strip headers"* and the *"strip data"*, respectively.

In the following, v_1^i and v_2^i denote the left and the right vertex of the entering edge of triangle i, counting i from 0. v_3^i denotes the remaining vertex of triangle i. The upper index is omitted if it is obvious which triangle is meant. If the winding of triangle $i-1$ is L, then $v_1^i = v_1^{i-1}$ and $v_2^i = v_3^{i-1}$,

if its winding is R, then $v_1^i = v_3^{i-1}$ and $v_2^i = v_2^{i-1}$. Furthermore, x_j^i, y_j^i, and h_j^i denote the x/y-coordinates and the height value of vertex v_j^i, and t^i denotes the type of triangle i.

By using this notation, the strip data entry specifying triangle i consists of the triangle type t^i and the height value h_3^i of the new vertex. Since the vertices of the entering edge of the very first triangle have to be stored explicitly, two additional entries containing the height values h_1^0 and h_2^0 are added to the beginning of the triangle list (their triangle type field is unused). Therefore, the entry of triangle i is located at index $i+2$, counting from 0. Figure 5 shows the layout of the strip data. Triangle types are always stored in 2 bits (values of 0-2 encode the triangle types A-C, while 3 is used to mark the end of the strip data), but the number of bits k used to quantize the height values varies from tile to tile. Since the strip entries are densely packed and no padding is used, bit arithmetic is necessary to read a single entry.

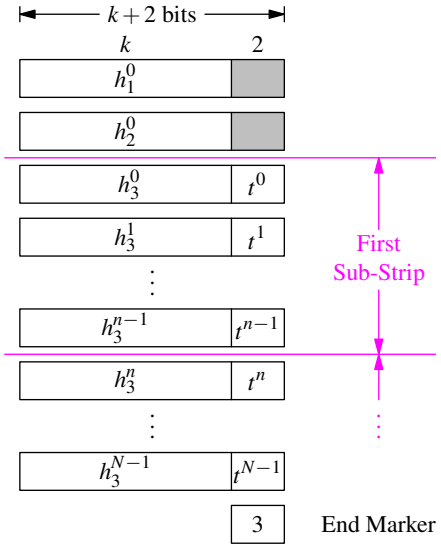


Figure 5: *Layout of the strip data. Each entry specifies a triangle by its type and the height value of the new vertex.*

Each strip header uses 32 bits to specify v_1, v_2, and the winding of the first triangle of the respective sub-strip. Figure 6 shows the layout of a single strip header. Since v_1 can never lie on the border of a tile, which can be shown by induction over the replacement system, and because the tile size in our implementation is 513×513, the x/y-coordinates of v_1 can be stored using 9 bits each. The position of v_2 is specified relative to v_1 by storing the orientation and the base 2 logarithm of their distance in the underlying 2D integer lattice (measured in the $\|\cdot\|_\infty$ norm). The orientation requires 3 bits (8 possible orientations S, SE, E, NE, N, NW, W and SW), and because the distance can take the values $2^0, 2^1, \ldots, 2^8$, its base 2 logarithm can be stored using 4 bits.

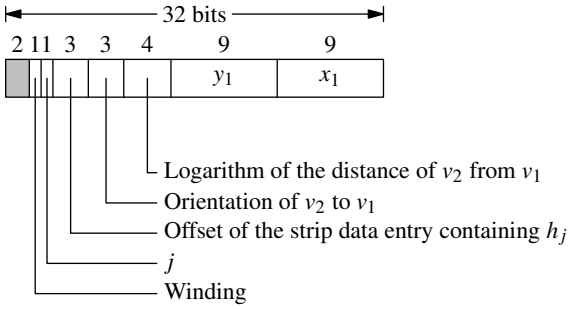


Figure 6: *Layout of a single strip header. A strip header specifies v_1, v_2 and the winding of the first triangle of a sub-strip.*

Due to the generalized strip representation of the triangle mesh, each of v_1 and v_2 of the first triangle of the sub-strip either corresponds to v_3 of one of the previous triangles in the strip or is equal to v_1 or v_2 of the very first triangle of the entire strip. The height values of v_1 and v_2 are thus contained in the data entries preceding the entry of the current triangle. We avoid a replication of these height values in the strip header by storing the (negative) offsets of the strip data entries containing these values relative to the entry of the current triangle i. In the following we show that these offsets can be stored using $3+1$ bits.

If the windings of the current triangle's predecessors $i-1$, $i-2, \ldots, i-(m-1)$ are L and the winding of triangle $i-m$ is R, then

$$\begin{aligned} v_1^i &= v_1^{i-1} = \cdots = v_1^{i-(m-1)} = v_3^{i-m} \qquad \text{and} \\ v_2^i &= v_3^{i-1}. \end{aligned}$$

Accordingly, if the windings of triangles $i-1, i-2, \ldots, i-(m-1)$ are R and the winding of triangle $i-m$ is L, then

$$\begin{aligned} v_1^i &= v_3^{i-1} \qquad \text{and} \\ v_2^i &= v_2^{i-1} = \cdots = v_2^{i-(m-1)} = v_3^{i-m}. \end{aligned}$$

Since the angles in every triangle are at least 45°, m is at most 7, otherwise the triangles would overlap. One of the offsets used to encode the height values is always 1, because one of v_1 or v_2 has already been stored as v_3 of the preceding triangle. The other offset references one of the 7 preceding triangles, but not the immediate predecessor. Since the very first triangle stores 3 instead of 1 height value, an offset ranging from 2 to 9 is required. We use 3 bits to store the offset different from 1 (decreased by 2) and 1 bit to store the index (1 or 2) of the respective vertex.

Finally, 1 bit is necessary to store the winding (L or R) of the first triangle of the sub-strip. The remaining 2 bits of the strip header are unused in the current implementation. In Figure 7 the compression of a restricted quadtree mesh using our approach is exemplified.

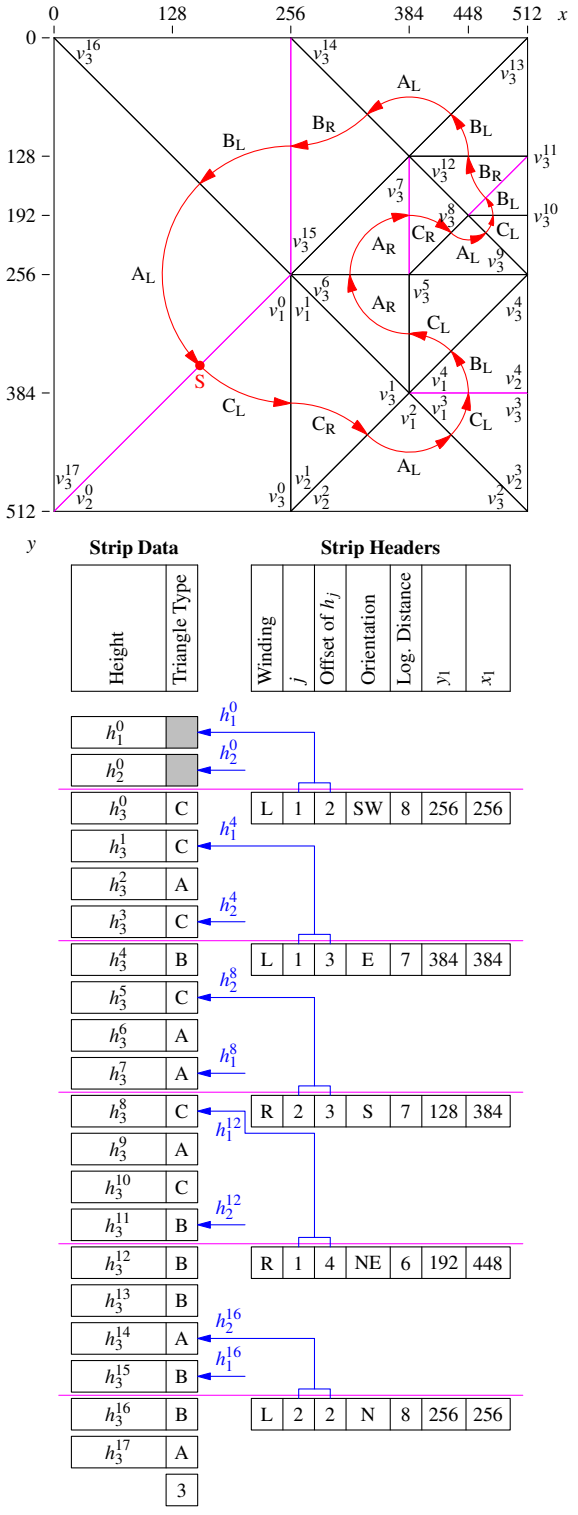


Figure 7: *The compression of a restricted quadtree mesh using our compression scheme is illustrated. To demonstrate the construction of the strip headers, a sub-strip length of $n = 4$ (instead of $n = 16$) is used in this example.*

5. GPU-Based Geometry Decoding

Geometry decoding is performed entirely on the GPU using geometry shaders as well as bit and integer arithmetic. In Direct3D 10 capable graphics hardware the geometry shader stage is placed directly after the vertex shader stage. In contrast to the vertex shader, the geometry shader takes as input an entire graphics primitive (e.g., a point or a triangle) and outputs zero to multiple new primitives. The geometry shader appends the new primitives to one or multiple output streams, which can be written back into graphics memory by using the stream output stage directly located after the geometry shader stage.

In the following, we present two implementations for decoding the geometry on the GPU, both returning an identical triangle list representation of the mesh. In the first implementation, the triangles of each sub-strip are generated in a single rendering pass using the geometry shader. In the second implementation, these triangles are created iteratively in multiple rendering passes using the fragment shader and a "ping-pong" technique, and they are then copied to the destination vertex buffer using the geometry shader. While the first implementation is straightforward, it results in considerably slower performance on current GPUs due to performance limitations imposed by the geometry shader stage.

5.1. Geometry Shader Implementation

In this section we show how the geometry can be decoded entirely on the GPU in a single rendering pass using the geometry shader. Once a tile is selected for decoding, the corresponding strip headers and the strip data are first loaded into a vertex buffer and a shader resource buffer, respectively. These buffers are reused for every tile. Then, a draw call is issued to render a point list as large as the number of strip headers, with the i^{th} strip header in the vertex buffer and a unique vertex id being associated as attributes to the i^{th} point. The id is automatically generated on the GPU and enumerates the points starting from 0. In this way, strip headers along with ids are fed into the graphics pipeline and are passed through the vertex shader stage into the geometry shader stage.

Using bit arithmetic, the geometry shader first decodes every incoming strip header to determine v_1 and v_2 of the first triangle of the respective sub-strip. The strip header specifies the x/y-coordinates of v_1 and v_2, as well as the offsets of the strip data entries containing their height values. The index of the strip data entries is computed from the associated id. If i denotes the strip id and Δ denotes the offset of the strip data entry containing v_1's or respectively v_2's height value, then the index is $2 + n \cdot i - \Delta$. To fetch the respective strip data entries, the number of bits used to quantize the height values of the tile is required (since the entries are densely packed and need to be extracted using bit arithmetic). This number is issued as a constant buffer variable.

The geometry shader now starts building the sub-strip successively. To construct a triangle, the respective strip data entry containing the triangle type and the height value of v_3 is fetched from the strip data buffer. The strip data entry is located at index $2+n\cdot i+j$, where j denotes the number of the current triangle within the sub-strip, counting from 0. In case of the first triangle, the winding is fetched from the strip header, otherwise it is determined from the triangle case of the previous triangle and the type of the current triangle. To do so, a GPU constant buffer storing the triangle case depending on these parameters serves as look-up table. Knowing v_1 and v_2 of the current triangle, as well as the triangle case, the x/y-coordinates of v_3 can be computed. By considering the winding of the current triangle, v_1 and v_2 of the next triangle are determined as described in Section 4.2. Finally, by using the stream output stage, the three vertices of the constructed triangle are written to a vertex buffer. Similar to [SW06], each vertex is encoded into 32 bits—10 bits for each of the x- and y-coordinate, and 12 bits for the height value—and is decoded on-the-fly during rendering in the vertex shader.

Note that by using a maximum of 12 bits for quantization and choosing a quantization error tolerance of $\varepsilon_Q := 0.25\,\varepsilon_\ell$ (see Section 4), the maximum height difference within each tile may be up to $2048\,\varepsilon_\ell$, with ε_ℓ denoting the world-space error tolerance of the respective level. Since in our implementation the tile extent is $512\,\varepsilon_\ell$, the maximum height difference can be up to 4 times larger than the tile extent. This corresponds to an average slope of $\arctan(4)\approx 76^\circ$ across the tile, which we did not encounter in any of our data sets.

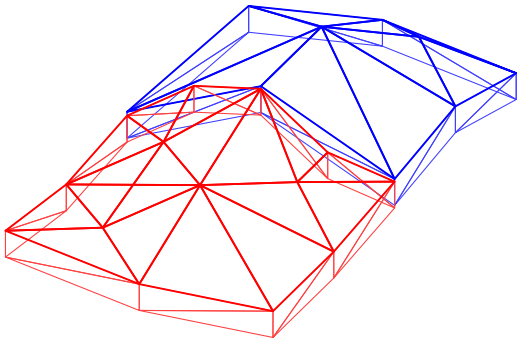


Figure 8: *Illustration of the skirts that are rendered around the border of each tile to hide cracks between adjacent tiles.*

Inherent to the the tile-based approach we use is the problem of cracks between adjacent tiles. Since each tile is meshed separately, T-vertices at tile boundaries can occur. Furthermore, because the height values in different tiles are quantized using different numbers of bits, at tile boundaries the same terrain sample might be encoded at a slightly different position. To avoid cracks we render skirts [Ulr02] around the border of each tile as depicted in Figure 8. The creation of these skirts is directly incorporated into the geometry decoding stage, by creating two additional triangles for each triangle having a border edge. To encode the height of the bottom vertices of the skirts in the packed 32 bit vertex format, we store the original height values and use a special mark to indicate that a height value has to be reduced by the height of the skirts during decoding.

5.2. Fragment Shader Implementation

In the second implementation, we use the fragment shader to successively create the triangles of each sub-strip in multiple rendering passes. In the j^{th} rendering pass, the j^{th} triangle of each sub-strip is generated. Since every triangle is dependent on the preceding triangle in the sub-strip, the result of each pass is needed as input to the next pass. This is implemented by using a "ping-pong" technique, which means that generated triangles are written into two textures that are alternately used as input and output. After all triangles have been constructed, they are finally copied to the destination vertex buffer using the geometry shader.

We use a four component unsigned integer texture format with 32 bits per component (R32G32B32A32) to store one triangle per texel. Using the packed 32 bit vertex format described in the previous section, the triangle's vertices fit into the RGB components. The remaining component is used to store the triangle case, which is needed to determine the winding of the next triangle in the sub-strip.

The mapping between texels and triangles is illustrated in figure 9, with Δ^i_j denoting triangle j of sub-strip i. In the second column, parallel decoding of multiple tiles is illustrated. This is done by associating with each tile a set of columns of the used textures. Each texture is divided into $\frac{n}{2}$ blocks, where the sub-strip length n is required to be even. In the current implementation, each block has a width and height of $w=4096$ and $h=8$ texels, respectively.

Decoding of a tile's geometry starts with loading the strip headers and the strip data into two shader resource buffers. Then, n ping-pong passes are performed to iteratively generate all triangles. In pass j, triangle j of each sub-strip is created. These triangles are written into block $\left\lfloor \frac{j}{2} \right\rfloor$ of texture j mod 2. The number of bits used to quantize the height values is issued as a vertex attribute, and it is thus available in the fragment shader.

Writing a triangle into the respective texel is performed in the fragment shader. First, the index i of the sub-strip containing this triangle is computed as $i=h\cdot u+v$, where u and v are the texel's coordinates within the *block*. Then, v_1 and v_2 are determined. In pass 0, which builds the first triangle of each sub-strip, the respective strip header needs to be fetched from the strip headers buffer at index i. The strip header is decoded to obtain the x/y-coordinates of v_1 and v_2, as well as the offsets of the strip data entries containing their height values. These entries are fetched from the strip data buffer at indices $2+n\cdot i-\Delta$, where Δ denotes the respective offset. In all other passes $j=1,\ldots,n-1$, the vertices v_1 and

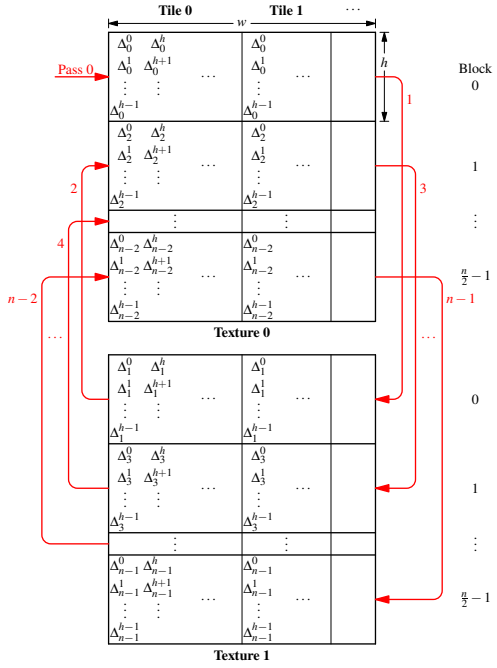


Figure 9: *This illustration shows the mapping between texels and triangles used in the fragment shader implementation. Based on a "ping-pong" technique, the triangles of each sub-strip are created iteratively in multiple rendering passes and are stored in two textures, which are alternately used as input and output.*

v_2 are determined from the previous triangle in the sub-strip as described in Section 4.2. This triangle has been generated in pass $j-1$, and it is thus stored in texture $(j-1) \bmod 2$, which is bound to the pipeline as a shader resource. The coordinates of the respective texel to be fetched are computed as $\left(u, h \cdot \left\lfloor \frac{j-1}{2} \right\rfloor + v\right)$.

After v_1 and v_2 are known, v_3 is computed in the same way as in the geometry shader implementation, with the triangle's strip data entry being located at index $2+n \cdot i+j$. The fragment shader finally outputs the vertices of the triangle along with the triangle case as a four component unsigned integer vector.

After n ping-pong passes, all triangles have been created. In an additional $(n+1)^{\text{st}}$ rendering pass, we copy the decoded geometry into the destination vertex buffer, which allows for a highly efficient rendering of the triangle list as proposed in the next section. In this pass we also create the skirts used to hide the cracks between adjacent tiles.

The copy pass starts by rendering a point list as large as $\left\lceil \frac{N}{2} \right\rceil$, with N denoting the number of triangles in the entire strip. The only attribute being associated to the points is the vertex id, which is implemented in Direct3D 10 by setting the input layout to NULL and adding the SV_VERTEXID semantic to the vertex shader input declaration. With each point two triangles are copied.

The triangles are read from the textures in the vertex shader. If m denotes the vertex id, then triangles $2m$ and $2m+1$ of the entire strip are read. These are triangles $(2m) \bmod n$ and $(2m+1) \bmod n$ of sub-strip $\left\lfloor \frac{2m}{n} \right\rfloor$, and are thus stored in texel $(\left\lfloor \frac{i}{h} \right\rfloor, h \cdot \frac{j}{2} + i \bmod h)$ of texture 0 and 1, with $i := \left\lfloor \frac{2m}{n} \right\rfloor$ and $j := (2m) \bmod n$. The vertex shader passes the two triangles to the geometry shader, which finally writes the triangles to the vertex buffer. In the destination vertex buffer the triangles are stored in the same order as they occur along the triangle strip. It can thus be exploited in the rendering pass that every triangle shares two vertices with its immediate predecessor in the strip.

6. Rendering

After geometry decoding, the quadtree mesh of a tile is available as a triangle list, with every vertex being encoded into 32 bits—10 bits for each of the x- and y-coordinate, and 12 bits for the height value. We render this list as an *indexed* triangle list, i.e., the 32 bit values are interpreted by the GPU as indices. Vertex attributes are not specified. In the vertex shader, the respective index value, and thus the encoded vertex, is accessed with the SV_VERTEXID semantic, and it is decoded using bit arithmetic.

By using indexed drawing, the GPU's vertex cache is utilized and caches the vertex shader output for the last few rendered indices. Especially in the current scenario, where each triangle shares two vertices with its predecessor, this cache is very effective. In particular, the vertex shader needs to be invoked only once for each triangle, resulting in almost twice the triangle throughput compared to rendering a non-indexed triangle list. On current graphics hardware, we achieve a rendering throughput of about 300 million triangles per second by using indexed drawing, compared to a throughput of about 170 million triangles per second using non-indexed drawing.

7. System Integration

In this section we outline the integration of the proposed geometry compression scheme into a terrain rendering system including LOD determination as well as CPU and GPU memory paging. In an off-line pre-processing step, the regular height map is encoded into a pyramidal data structure as described in Section 4. A photo texture, if available, is processed in the same way, except that the texture is compressed using the S3TC scheme. Each tile thus consists of a triangle mesh and a texture. The tiles are organized as a quadtree, which we refer to as the tile tree.

In each frame, the set of tiles used to render the terrain at the current view is determined. This is done by selecting

tiles as coarse as possible, yet resulting in a screen-space error below a given error threshold τ. This error is measured in pixels. To determine these tiles, we use the screen-space error metric proposed by Cohen [Coh98], which is based on the theorem on intersecting lines. To not exceed τ, the world space error tolerance $\varepsilon(P)$ at a point P with depth P_z to the camera is computed as

$$\varepsilon(P) = \underbrace{\frac{2}{h} \cdot \tan\left(\frac{\theta}{2}\right) \cdot \tau}_{=:\alpha} \cdot P_z, \tag{2}$$

where h denotes the height of the view port in pixels and θ the vertical field of view. Since this metric is strictly monotonic and increasing with depth, it only has to be evaluated at the point P' of the tile's bounding box with the least depth to the camera. According to Equation 1, a tile at level ℓ is used for rendering, iff $\varepsilon_\ell \leq \varepsilon(P') < \varepsilon_{\ell+1}$. Then, the tile has a screen-space error of at most τ over its entire domain.

The set of tiles used to render the terrain changes with the movements of the viewer, i.e., a tile is replaced by its children if it has become too coarse, and vice versa. The geometric difference between a tile and its children at consecutive levels ℓ and $\ell-1$ is limited by $\varepsilon_\ell + \varepsilon_{\ell-1} = \frac{3}{2}\varepsilon_\ell$ (considering that $\varepsilon_\ell = 2\varepsilon_{\ell-1}$), which results in a screen-space difference of up to $\frac{3}{2}\tau$. Consequently, to avoid visible level changes τ is set to $\frac{2}{3}$ instead of 1 pixel.

To compute the world-space height of the skirts needed to hide T-vertices and quantization artifacts, we observe that adjacent tiles can differ by at most $\lfloor \log_2((M-1)\alpha+2) \rfloor$ levels, with M denoting the tile size in samples and α being defined in Equation 2 (for a derivation please refer to the appendix). The geometric difference on the shared border of a tile at level ℓ and an adjacent tile at level $\ell+\Delta\ell$, $\Delta\ell \in \mathbb{Z}$, is limited by $\varepsilon_\ell + \varepsilon_{\ell+\Delta\ell} = \left(2^{\Delta\ell}+1\right)\varepsilon_\ell$. Since for typical camera settings the LOD difference is at most 1, for a tile at level ℓ skirts of height $3\varepsilon_\ell$ are required. In screen-space, the height of the skirts is limited by 2τ, which is $\frac{4}{3}$ pixels in our implementation. By using skirts to fill the cracks between adjacent tiles, visual artifacts along tile borders can be avoided entirely.

7.1. Disk Access

To reduce the number of disk seek operations, 4×4 tiles in each level are packed together and stored as a single data block on disk. We refer to these groups as pages, and these pages are always loaded into main memory as a whole. The set of pages inherits the tree structure from the tile tree, as depicted in Figure 10. This data structure, which we refer to as the page tree, is stored on disk. During runtime, the page tree is dynamically loaded into main memory, as required due to the movements of the viewer. The tile tree is built and destroyed simultaneously with the page tree, i.e., when a page is added to or removed from the page tree, the tiles contained in this page are added to or removed from the tile tree. Note that the tiles only store pointers to the respective data blocks in the page tree, so that the data is not stored twice in main memory.

Both trees are built top-down and destroyed bottom-up. To further reduce the number of disk seek operations, each page is accompanied by the information that is required to access its children, i.e., the position of each child page in the data file, its size and bounding box.

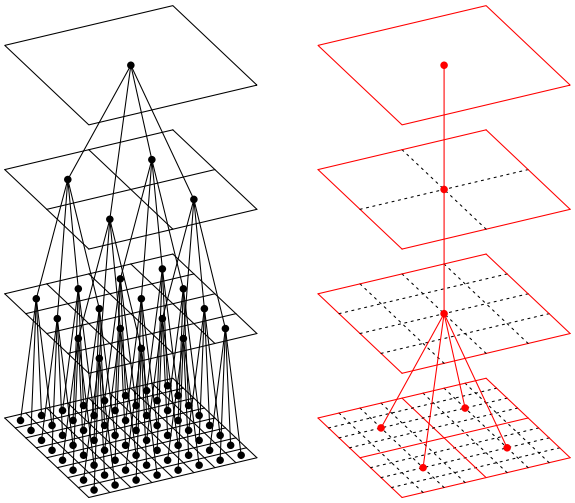


Figure 10: *Illustration of the tile tree (left) and the corresponding page tree (right), which is generated by grouping 4×4 tiles of each level into pages.*

To be able to hide disk access latencies, we have realized a software scheme to pre-fetch pages that might be required in upcoming frames. We employ a sphere-shaped pre-fetching region, i.e., the region that is tried to be loaded ahead of time. The advantage of a sphere-shaped region is that it supports arbitrary rotations. As has been shown by Ng et al. [NNT*05], although the use of a sphere-shaped region increases the memory requirement by a factor of two, when moving along the line of sight the amount of data to be read from disk is only marginally increased (see Figure 11).

At each level in the data hierarchy we use a pre-fetching region with different radius. To determine the extent of this region for a particular LOD, we exploit the fact that the LOD metric we use behaves linearly with the z-coordinate in camera space. Consequently, we can directly compute the radius of the pre-fetching region at a certain level such that the region contains all points in the view frustum at this LOD.

Page loads are handled asynchronously by a separate IO-thread that works independently of the rendering thread. In every frame, those pages are determined whose bounding boxes have just entered or left the pre-fetching region due to the movement of the viewer. For all entering pages a request

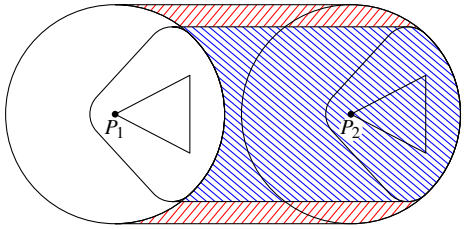


Figure 11: *Illustration of the amount of data that has to be fetched from disk when moving from P_1 to P_2 along the line of sight using differently shaped pre-fetching regions. For the cone shape, the data in the blue area has to be loaded. For the sphere shape, the data in the red area has to be loaded in addition.*

is created and added to a request queue. Exiting pages are removed from this queue. The IO-thread processes the queue in a particular order and loads the requested pages. The processing order is determined by giving pages at a coarser LOD a higher priority, and by preferring of all pages at the same LOD those in the center of the field of view. In this way, the visual importance of rendered tiles is taken into account. Exiting pages are not disposed immediately, but they are stored in an LRU page cache of limited size.

7.2. CPU-GPU Streaming

In every frame, the tile tree currently available in main memory is traversed in preorder, and the tiles to be rendered are determined by view frustum culling and LOD computation. To exploit frame-to-frame coherence only tiles not already residing in graphics memory are uploaded to the GPU. For each of these tiles, we first build a mipmap pyramid by gathering the mipmap levels from the tile hierarchy on-the-fly. This mipmap pyramid as well as the tile's compressed geometry is then uploaded to the GPU, where the geometry is decoded instantaneously (using the fragment shader based implementation described in Section 5.2).

To avoid time-consuming creation and disposal of resources in graphics memory, we create a resource pool once at the beginning and reuse these resources over and over again. While the texture size is constant for all tiles, the size of the decoded geometry can vary significantly between different tiles. To flexibly provide vertex buffers of appropriate sizes, we create vertex buffers of the maximum required size and use a buddy system [Knu97] to manage allocation of individual blocks within these buffers.

8. Results and Analysis

To validate our approach, we have integrated the geometry compression method into a terrain rendering engine. The renderer streams compressed data from a LOD hierarchy with a page tree. Optionally, orthographic aerial photography can be supplied as DXT1-compressed texture tiles. To measure performance, we run several benchmarks on a standard PC with an Intel Core2Duo E6600 processor, 3 GB of RAM, an NVIDIA GeForce 8800GTX with 768 MB of video memory, and, unless noted otherwise, a single Samsung 500 GB hard disk. For all tests, the far plane was set to 600 km and the screen-space error tolerance was set to $\frac{2}{3}$ pixels.

8.1. Data Sets & Pre-Processing

Two different DEMs were used. Our first data set is a DEM of the entire USA at a resolution of 10 m $\times$ 10 m obtained from the U.S. Geological Survey [USG] and corresponds to a 637 K $\times$ 281 K uniform grid. In a pre-processing step we synthesized a texture storing 1 texel per 10 m $\times$ 10 m. The size of the input data is about 333 GB for the geometry (at 16 bits per sample) and 500 GB for the synthetic texture. After meshing and geometry compression using a world-space error tolerance of 5 m (2.5 m) at the finest level, 3.9 GB (10.5 GB) are required to store the geometry. This results in an overall compression rate of 85:1 (32:1) compared to the DEM size. This data set compresses exceptionally well due to large planar water surfaces. Our compression scheme allocates 11.0 bits (11.5 bits) per triangle on average, of which 2 bits are used to encode the topology (i.e., the triangle type) and an average of 2 bits per triangle are occupied by the strip header. The remaining 7.0 bits (7.5 bits) encode the height value. When compared to the packed 32 bit vertex format described in Section 5.1 that allocates 96 bits per triangle, we achieve a compression ratio of 8.7:1 (8.4:1). The S3TC compressed photo texture requires 111 GB, including mipmaps.

Our second data set is a DEM of the State of Utah at a resolution of 5 m $\times$ 5 m, accompanied by an orthographic photo texture at a resolution of 1 texel per 1 m $\times$ 1 m. This amounts to a total of 92 K $\times$ 120 K samples for the geometry, and 460 K $\times$ 600 K samples for the texture, resulting in 20.6 GB and 772 GB to store geometry and texture, respectively. The data was provided by the State of Utah through the Utah GIS Portal [Sta]. After pre-processing using a world-space error tolerance of 2.5 m (1.25 m) at the finest level, the geometry was reduced to 1.6 GB (4.2 GB), resulting in a compression rate of about 13:1 (5:1) compared to the DEM size. Our compression scheme allocates 12.0 bits (12.3 bits) per triangle on average. Again, 4 bits are occupied by topology and amortized strip header. The compression ratio when compared to 96 bits per triangle is 8.0:1 (7.8:1). The photo texture was compressed to 171 GB, including mipmaps.

Pre-processing of the geometry typically takes less than 30 minutes for a heightfield of the size of the Utah data set (11 G samples) on a single core PC. For the texture, preprocessing is typically dominated by DXT1 encoding times. Using a multi-threaded wrapper of the publicly available, highly optimized Squish library [Bro], compressing a texture of the size of the Utah data set (270 G samples) using

the iterative cluster-fit algorithm takes around 3 days on a quad-core system.

For the following tests, world-space error tolerances of 2.5 m and 1.25 m at the finest level were used for the USA and the Utah data set, respectively.

8.2. Geometry Decoding

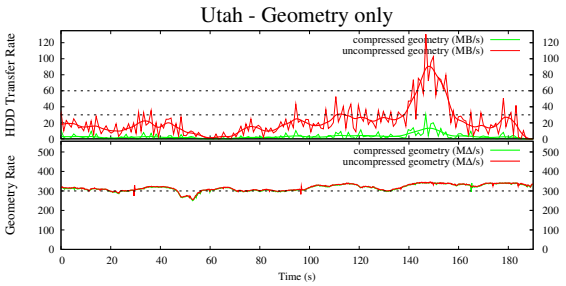


Figure 12: *Recorded flight over Utah, resolved at a spacing of 5 m without texture. Top: HDD transfer rate in MB/s. Bottom: Geometry rate in million triangles per second. The camera velocity was set to constant 14,000 km/h.*

To measure the impact of our geometry compression scheme we used the Utah data set without texture. To avoid the hard disk becoming the limiting factor, we used a striped RAID 0 delivering a sustained rate of about 200 MB/s. Thus all data necessary to keep the screen-space error below $\frac{2}{3}$ pixels on a 2560×1600 view port was being loaded during a flight at a constant camera velocity of 14,000 km/h. As can be seen in the upper diagram of Figure 12, in regions with high data densities the amount of data required to maintain the screen-space error tolerance can increase significantly. Here, the red curve corresponds to rendering from uncompressed geometry, i.e., from the 96 bits per triangle format, while the green curve employs our data compression scheme followed by GPU-based decoding. The thinner curves show actual measurement data, while the bold curves show the general trend. Note that due to our pre-fetching scheme the inability to deliver a single peak of the thinner curves does not automatically incur a lack of detail. In contrast, if we were not able to deliver 100 MB/s sustained rate at time index 150 s, a clearly visible lack of detail would occur. Current hard disks are capable of delivering sustained rates of around 90-100 MB/s (due to the novel perpendicular recording technology), with 50-60 MB/s still being the standard. Hence, sustained rates of more than 60 MB/s currently imply novel hardware or RAIDs. The compressed geometry, however, can be fully streamed at rates of as low as about 15-18 MB/s, meaning that even notebooks are able to deliver full visual quality. Note that this situation is even exacerbated in case of textured terrain.

The lower diagram in Figure 12 shows the triangle rendering throughput for both compressed and uncompressed geometry. As can be seen, they coincide almost perfectly at a sustained rate of about 300 million triangles per second. Note that this number includes the triangles used to render skirts, which amount to about 5% of the total number of triangles rendered. Since in both cases (compressed and uncompressed) the same amount of geometry has to be rendered, we conclude that the GPU-based decoding comes virtually for free in practical systems exploiting frame-to-frame coherence.

We also measured the data throughput from main memory to video memory. For this test, we compare the upload of uncompressed data with the upload and decoding of compressed data. For comparison reasons, we measure all compressed data by its uncompressed size (in packed 32 bit vertex format). For uncompressed data, we achieve a throughput of about 1.5 GB/s, while uploading and decoding on the GPU achieves over 2.5 GB/s. Furthermore, compressed data can be decoded at about 400 MB/s on the CPU, while the GPU is able to decode more than 2.5 GB/s. Note that in this benchmark we are currently limited by the GPU-based decoder, and we expect the throughput of compressed data to increase even further with future hardware.

8.3. Video Memory Consumption

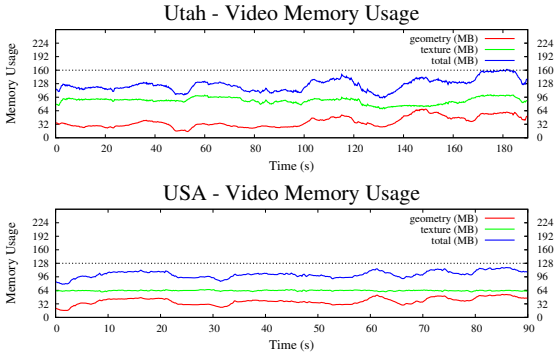


Figure 13: *Video memory usage for the Utah and USA data sets. Both data sets were rendered to a 2560×1600 view port at a screen-space error tolerance of $\frac{2}{3}$ pixels.*

In Figure 13 we illustrate the amount of video memory occupied by our method during flights over each of the two data sets. Again, measurements of the memory occupied by geometry refer to the packed 32 bit vertex format, and the texture is stored in DXT1 format. As can be seen in Figure 13, the memory usage for the Utah data set is always below 160 MB, even though the view port was 2560×1600 pixels. For the USA data set, video memory usage is always below 128 MB. The difference in video memory usage stems from the fact that the USA data set has not enough resolution for the relatively low altitude of the flight. For the Utah data set we observed no lack in resolution, i.e., the LOD rendered was always sufficient to guarantee the prescribed pixel tolerance. Consequently, the measurements for Utah provide a

good estimate of the typical video memory usage of our system.

8.4. Textured Terrain

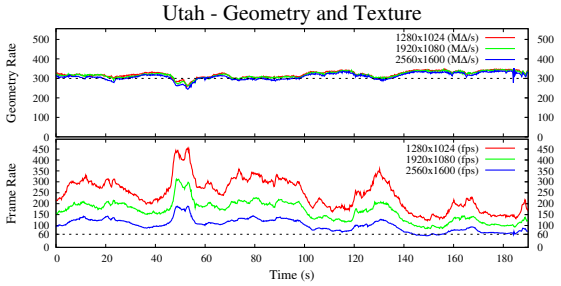


Figure 14: *Recorded flight over Utah demonstrating the effect of various display resolutions. The diagrams show triangle throughput and frame rate over time.*

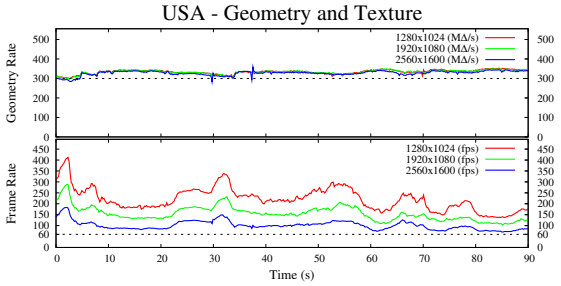


Figure 15: *Recorded flight over the USA demonstrating the effect of various display resolutions. The diagrams show triangle throughput and frame rate over time.*

Finally, we demonstrate the frame rate that can typically be achieved for textured terrain in dependence of view port resolution. In both Figure 14 and Figure 15, red corresponds to a resolution of 1280 × 1024 (1.25 megapixels), green corresponds to 1920 × 1080 (full HDTV resolution at 2 megapixels), and blue corresponds to 2560 × 1600 (4 megapixels). In all cases the sustained triangle throughput is about 300 million triangles per second, indicating that we are strictly limited by the GPU's triangle throughput. The frame rates stay well above 60 fps almost everywhere at the highest resolution of 2560 × 1600. Also, the achieved frame rates scale better than inversely linear in the amount of pixels rendered, i.e., doubling the amount of pixels to be rendered does not automatically cut the frame rate in half. Considering that the triangle rate is almost constant, this indicates that our LOD scheme scales the amount of triangles slightly sub-linear in the size of the view port. The peak amount of triangles per frame is about 5.3 million triangles at the end of the Utah flyover and is caused by the highly detailed, non-planar landscape.

These benchmarks prove our method to be of significant practical value, since we can deliver high sustained frame rates as well as high geometry rates for very high view port resolutions. At the same time, disk traffic is minimized, which is a crucial feature when dealing with data sets that are several hundreds of gigabytes in size. Last but not least, the small footprint in video memory allows our algorithm to be integrated into more complex GIS systems that increase memory requirements due to additional data to be visualized.

9. Conclusion & Future Work

In this paper, we have presented a geometry compression scheme for restricted quadtree meshes and have demonstrated the use of this scheme in terrain rendering. The proposed scheme offers good compression rates as well as precise error control, and it allows the compressed data to be efficiently decoded on the GPU. In this way, bandwidth requirements in terrain rendering can be reduced significantly, both with respect to data transfer from disk to main memory and from main memory to GPU memory. In contrast to previous approaches like geometry clipmaps our method achieves lower compression rates, but we can guarantee a very small error tolerance using significantly fewer triangles.

Due to the smaller memory footprint of compressed DEMs, significantly more geometry can be cached in main memory. By using the GPU for geometry decoding, uploading and decoding of the compressed data is still about a factor of two faster than uploading uncompressed data from this memory. GPU decoding itself is about a factor of 6 faster than CPU decoding. By using the GPU for decoding, the CPU is free for other tasks like pre-fetching and data management. We validated these statements by integrating the compression and decoding schemes into a terrain rendering system, and we showed that high visual quality on high resolution displays is possible at interactive frame rates.

In the future, we will investigate on-the-fly decoding of compressed DEMs on the GPU to avoid multi-pass rendering and storing decoded DEMs in GPU memory. One additional future avenue of research is the design of prediction strategies to incrementally encode the height values along the path through the quadtree mesh, and thus to further reduce the amount of bits required to store these values.

References

[ASU86] AHO A., SETHI R., ULLMAN J.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Blo00] BLOW J.: Terrain rendering at high levels of detail. In *Proc. Game Developer's Conference* (2000).

[Bro] BROWN S.: Squish – DXT Compression Library. http://www.sjbrown.co.uk/?code=squish.

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

[CGG*03b] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.: Planet-sized batched dynamic adaptive meshes (P-BDAM). In *Proc. IEEE Visualization* (2003), pp. 147–154.

[CKS03] CORRÊA W. T., KLOSOWSKI J. T., SILVA C. T.: Visibility-based prefetching for interactive out-of-core rendering. In *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 1–8.

[Coh98] COHEN J. D.: *Appearance-Preserving Simplification of Polygonal Models*. PhD thesis, University of North Carolina at Chapel Hill, 1998.

[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 *Proc. IEEE Visualization* (1997), pp. 81–88.

[FEKR90] FERGUSON R. L., ECONOMY R., KELLY W. A., RAMOS P. P.: Continuous terrain level of detail for visual simulation. In *Proc. IMAGE V* (1990), pp. 144–151.

[FL79] FOWLER R. J., LITTLE J. J.: Automatic extraction of irregular network digital terrain models. In *Proc. ACM SIGGRAPH* (1979), pp. 199–207.

[Ger03] GERSTNER T.: Multiresolution visualization and compression of global topographic data. *GeoInformatica 7*, 1 (2003), 7–32.

[GGS95] GROSS M. H., GATTI R., STAADT O.: Fast multiresolution surface meshing. In *Proc. IEEE Visualization* (1995), pp. 135–142.

[GH95] GARLAND M., HECKBERT P. S.: *Fast Polygonal Approximation of Terrains and Height Fields*. Tech. Rep. CMU-CS-95-181, Carnegie Mellon University, 1995.

[GMC*06] GOBBETTI E., MARTON F., CIGNONI P., DI BENEDETTO M., GANOVELLI F.: C-BDAM – compressed batched dynamic adaptive meshes for terrain rendering. *Computer Graphics Forum 25*, 3 (2006), 333–342.

[HDJ04] HWA L. M., DUCHAINEAU M. A., JOY K. I.: Adaptive 4-8 texture hierarchies. In *Proc. IEEE Visualization* (2004), pp. 219–226.

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

[KLR*95] KOLLER D., LINDSTROM P., RIBARSKY W., HODGES L. F., FAUST N., TURNER G.: Virtual GIS: A real-time 3D geographic information system. In *Proc. IEEE Visualization* (1995), pp. 94–100.

[Knu97] KNUTH D. E.: *Fundamental Algorithms*, 3 ed., vol. 1 of *The Art of Computer Programming*. Addison-Wesley, 1997.

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

[LH04] LOSASSO F., HOPPE H.: Geometry clipmaps: Terrain rendering using nested regular grids. In *Proc. ACM SIGGRAPH* (2004), pp. 769–776.

[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 *Proc. ACM SIGGRAPH* (1996), pp. 109–118.

[LP01] LINDSTROM P., PASCUCCI V.: Visualization of large terrains made easy. In *Proc. IEEE Visualization* (2001), pp. 363–370.

[LP02] LINDSTROM P., PASCUCCI V.: Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE TVCG 8*, 3 (2002), 239–254.

[NNT*05] NG C.-M., NGUYEN C.-T., TRAN D.-N., TAN T.-S., YEOW S.-W.: Analyzing pre-fetching in large-scale visual simulation. In *Proc. Computer Graphics International* (2005), pp. 100–107.

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

[PD04] PLATINGS M., DAY A. M.: Compression of large-scale terrain data for real-time visualization using a tiled quad tree. *Computer Graphics Forum 23*, 4 (2004), 741–759.

[PFL78] PEUCKER T. K., FOWLER R. J., LITTLE J. J.: The triangulated irregular network. In *Proc. ASP-ACSM Symposium on DTM's* (1978).

[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, Center for Image Processing and Integrated Computing, University of California, Davis, 2000.

[RHSS98] RÖTTGER S., HEIDRICH W., SLUSALLEK P., SEIDEL H.-P.: Real-time generation of continuous levels of detail for height fields. In *Proc. WSCG* (1998), pp. 315–322.

[RLIB99] REDDY M., LECLERC Y., IVERSON L., BLETTER N.: TerraVision II: Visualizing massive terrain databases in VRML. *IEEE Computer Graphics and Applications 19*, 2 (1999), 30–38.

[S3] S3 INCORPORATED: Fixed-rate block-based image compression with inferred pixel values. US Patent 6658146.

[SN95] SUTER M., NÜESCH D.: Automated generation of visual simulation databases using remote sensing and GIS. In *Proc. IEEE Visualization* (1995), pp. 86–93.

[Sta] STATE OF UTAH: Utah GIS Portal. http://gis.utah.gov.

[SW06] SCHNEIDER J., WESTERMANN R.: GPU-friendly high-quality terrain rendering. *Journal of WSCG 14*, 1-3 (2006), 49–56.

[TG98] TOUMA C., GOTSMAN C.: Triangle mesh compression. In *Proc. Graphics Interface* (1998), pp. 26–34.

[TR98] TAUBIN G., ROSSIGNAC J.: Geometric compression through topological surgery. *ACM TOG 17*, 2 (1998), 84–115.

[Ulr02] ULRICH T.: Rendering massive terrains using chunked level of detail control. ACM SIGGRAPH Course "Super-size it! Scaling up to Massive Virtual Worlds", 2002.

[USG] USGS: U.S. Geological Survey. http://www.usgs.gov.

[VB87] VON HERZEN B., BARR A. H.: Accurate triangulations of deformed, intersecting surfaces. In *Proc. ACM SIGGRAPH* (1987), pp. 103–110.

[WMD*04] WAHL R., MASSING M., DEGENER P., GUTHE M., KLEIN R.: Scalable compression and rendering of textured terrain data. *Journal of WSCG 12*, 1-3 (2004), 521–528.

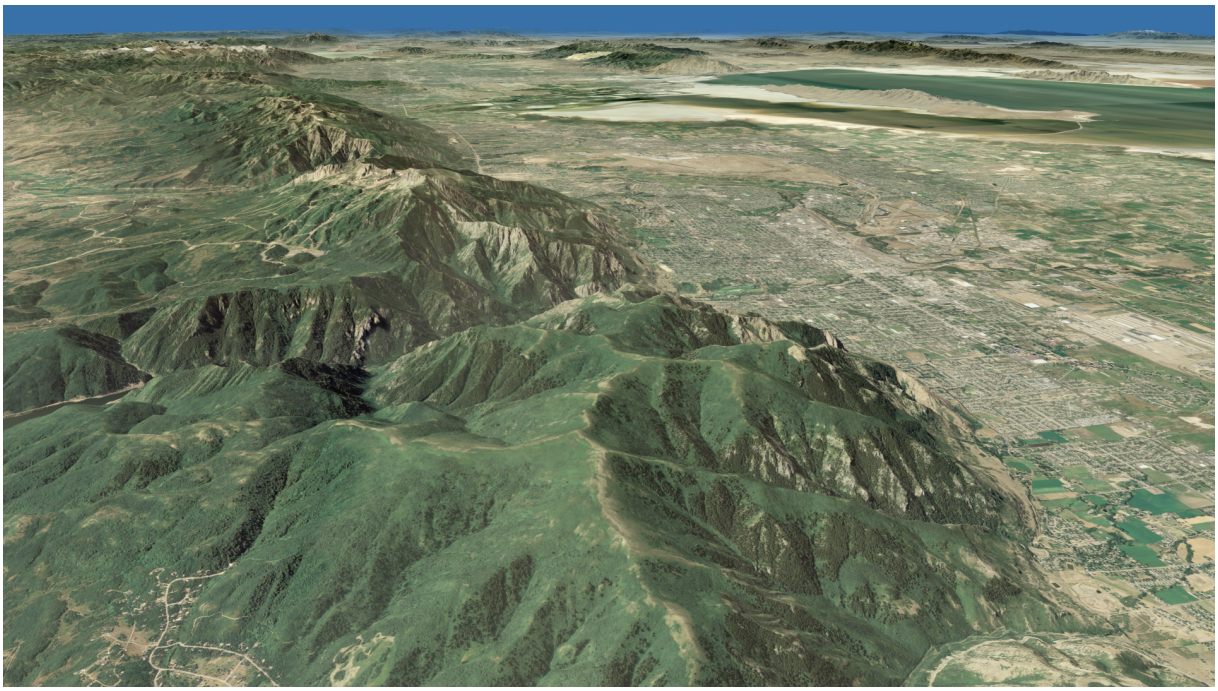


Figure 16: *View over Salt Lake City rendered on a 1920 × 1080 view port at a screen-space error of below one pixel. The data set comprises the entire State of Utah at a spacing of 5 m and 1 m for geometry and texture, respectively.*

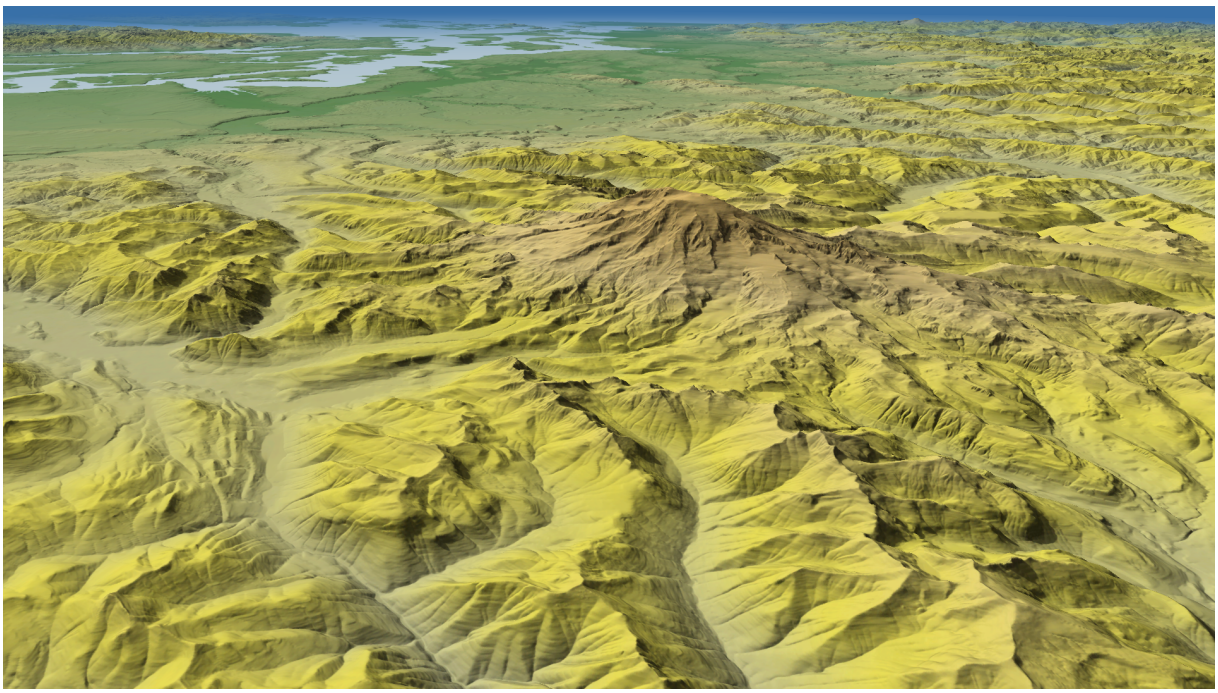


Figure 17: *View over Puget Sound rendered on a 1920 × 1080 view port at a screen-space error of below one pixel. The data set comprises the entire USA at a spacing of 10 m for both geometry and texture.*

APPENDIX

Height of Skirts

In this section we will derive the formula describing the minimum height of flanges that guarantees visually crack-free renderings.

A single tile at level ℓ consists of $M \times M$ height samples with a spacing equal to the level's world-space error tolerance ε_ℓ, so that the tile extent is $(M-1)\,\varepsilon_\ell$ (see Section 4). Note that in the following we consider only the horizontal extent of a tile. Further we use the error metric described in Section 7, with $\varepsilon(z) = \alpha \cdot z$, where α is only dependent on camera and view port parameters as well as the screen-space error tolerance. A tile at level ℓ is chosen for rendering, iff

$$\varepsilon_\ell \leq \varepsilon(z') < \varepsilon_{\ell+1}.$$

Here, z' is the the minimum depth of all bounding box points of the tile with respect to the camera.

As observed before, $\varepsilon_\ell = 2^\ell \varepsilon_0$. Obviously, the depth range $[z_\ell, z_{\ell+1}[$ in which level ℓ is used starts at z_ℓ with $\varepsilon(z_\ell) = \alpha \cdot z_\ell = \varepsilon_\ell$. Thus, $z_\ell = \frac{1}{\alpha}\varepsilon_\ell$. The range ends at $z_{\ell+1}$, for which analogously $z_{\ell+1} = \frac{1}{\alpha}\varepsilon_{\ell+1} = \frac{1}{\alpha}2\varepsilon_\ell = 2z_\ell$. Hence, the total length of the depth range in which level ℓ is to be applied is exactly z_ℓ.

Assume we have two *adjacent* tiles at levels ℓ and $\ell+\Delta\ell$, $\Delta\ell \in \mathbb{N}_0$, with minimum depths z' and z''. Since z' and z'' lie within the depth ranges $[z_\ell, z_{\ell+1}[$ and $[z_{\ell+\Delta\ell}, z_{\ell+\Delta\ell+1}[$, respectively, it is

$$\begin{aligned} |z' - z''| \geq z_{\ell+1} + \cdots + z_{\ell+\Delta\ell-1} &= \sum_{i=1}^{\Delta\ell-1} 2^i z_\ell \\ &\geq (2^{\Delta\ell} - 2)\, z_\ell \end{aligned} \tag{3}$$

(considering that the length of the depth range $[z_k, z_{k+1}[$ is z_k).

Since adjacent tiles intersect on an edge, it is

$$|z' - z''| \leq (M-1)\,\varepsilon_\ell. \tag{4}$$

Equation 3 and 4 yield

$$\begin{aligned} & \left(2^{\Delta\ell} - 2\right) z_\ell \leq (M-1)\,\varepsilon_\ell \\ \Leftrightarrow \quad & \left(2^{\Delta\ell} - 2\right) \frac{1}{\alpha} \leq M - 1 \\ \Leftrightarrow \quad & \Delta\ell \leq \log_2\left((M-1)\,\alpha + 2\right), \end{aligned}$$

and thus the level difference between adjacent tiles is limited by $L := \lfloor \log_2((M-1)\,\alpha + 2) \rfloor$.

Hence, the maximum geometric difference on the border between a tile at level ℓ and an adjacent tile is at most

$$\varepsilon_\ell + \varepsilon_{\ell+L} = \left(2^L + 1\right)\varepsilon_\ell.$$

We extrude skirts exactly by this amount downwards in our system. Whenever camera parameters such as field of view or view port dimensions are changed, L is re-evaluated.

Replacement System

In this section we will show for the replacement system used for geometry compression that the winding of each triangle can be determined from the type and winding of the previous triangle in the strip and the type of the current triangle.

Table 1: *System of inequalities and result of the fixpoint iteration for First (top) and Follow (bottom) sets. ε denotes the empty word.*

System of inequalities	Result
$\mathrm{First}(\mathrm{A_L}) \supseteq \{\mathrm{A_L}\} \cup \mathrm{First}(\mathrm{C_L})$	$\{\mathrm{A_L}, \mathrm{C_L}\}$
$\mathrm{First}(\mathrm{A_R}) \supseteq \{\mathrm{A_R}\} \cup \mathrm{First}(\mathrm{C_R})$	$\{\mathrm{A_R}, \mathrm{C_R}\}$
$\mathrm{First}(\mathrm{B_L}) \supseteq \{\mathrm{B_L}\} \cup \mathrm{First}(\mathrm{C_R})$	$\{\mathrm{A_R}, \mathrm{B_L}, \mathrm{C_R}\}$
$\mathrm{First}(\mathrm{B_R}) \supseteq \{\mathrm{B_R}\} \cup \mathrm{First}(\mathrm{C_L})$	$\{\mathrm{A_L}, \mathrm{B_R}, \mathrm{C_L}\}$
$\mathrm{First}(\mathrm{C_L}) \supseteq \{\mathrm{C_L}\} \cup \mathrm{First}(\mathrm{A_L})$	$\{\mathrm{A_L}, \mathrm{C_L}\}$
$\mathrm{First}(\mathrm{C_R}) \supseteq \{\mathrm{C_R}\} \cup \mathrm{First}(\mathrm{A_R})$	$\{\mathrm{A_R}, \mathrm{C_R}\}$

System of inequalities	Result
$\mathrm{Follow}(\mathrm{A_L}) \supseteq \{\mathrm{A_L}, \varepsilon\} \cup \mathrm{Follow}(\mathrm{B_L}) \cup \mathrm{First}(\mathrm{B_R})$	$\{\mathrm{A_L}, \mathrm{B_R}, \mathrm{C_L}, \varepsilon\}$
$\mathrm{Follow}(\mathrm{A_R}) \supseteq \mathrm{Follow}(\mathrm{B_R}) \cup \mathrm{First}(\mathrm{B_L})$	$\{\mathrm{A_R}, \mathrm{B_L}, \mathrm{C_R}\}$
$\mathrm{Follow}(\mathrm{B_L}) \supseteq \mathrm{Follow}(\mathrm{A_L}) \cup \mathrm{Follow}(\mathrm{C_R})$	$\{\mathrm{A_L}, \mathrm{B_R}, \mathrm{C_L}, \varepsilon\}$
$\mathrm{Follow}(\mathrm{B_R}) \supseteq \mathrm{Follow}(\mathrm{A_R}) \cup \mathrm{Follow}(\mathrm{C_L})$	$\{\mathrm{A_R}, \mathrm{B_L}, \mathrm{C_R}\}$
$\mathrm{Follow}(\mathrm{C_L}) \supseteq \mathrm{First}(\mathrm{B_L}) \cup \mathrm{First}(\mathrm{A_R})$	$\{\mathrm{A_R}, \mathrm{B_L}, \mathrm{C_R}\}$
$\mathrm{Follow}(\mathrm{C_R}) \supseteq \mathrm{First}(\mathrm{B_R}) \cup \mathrm{First}(\mathrm{A_L})$	$\{\mathrm{A_L}, \mathrm{B_R}, \mathrm{C_L}\}$

In our proof, we use concepts of formal language theory, i.e., grammars as well as First and Follow sets. Formally, a grammar G consists of three finite sets V, Σ, P, and a so-called *starting symbol* $S \in V$. V contains *variables*, Σ is the alphabet containing the *terminals* or symbols used to form *words* $\omega \in \Sigma^*$, and P contains replacement rules called *productions*. For a more thorough introduction to these concepts, we refer to [ASU86].

For sake of simplicity, we will slightly adapt the definition of First and Follow sets to operate on a replacement system that does not distinguish between variables and terminals. In this way, unnecessary productions are avoided.

Let $\Sigma := \{\mathrm{A_L}, \mathrm{A_R}, \mathrm{B_L}, \mathrm{B_R}, \mathrm{C_L}, \mathrm{C_R}\}$. Further, let P be defined as follows (see Section 4.1).

$$\begin{aligned} P := \{ \quad & \mathrm{A_L} \to \mathrm{C_L B_L}, \quad && \mathrm{A_R} \to \mathrm{C_R B_R}, \\ & \mathrm{B_L} \to \mathrm{C_R A_L}, && \mathrm{B_R} \to \mathrm{C_L A_R}, \\ & \mathrm{C_L} \to \mathrm{A_L B_R}, && \mathrm{C_R} \to \mathrm{A_R B_L} \ \} \end{aligned}$$

Then, beginning with $S := \mathrm{A_L A_L A_L A_L}$, the word corresponding to the triangle strip built during meshing is obtained by successively applying productions.

The First and Follow sets are defined as

$$\begin{aligned} \mathrm{First}(a) &:= \{b \in \Sigma : \exists \omega \in \Sigma^* : a \to^* b\omega\}, \\ \mathrm{Follow}(a) &:= \{b \in \Sigma : \exists \omega_1, \omega_2 \in \Sigma^* : S \to^* \omega_1 a b \omega_2\}. \end{aligned}$$

Here, $\to^*$ denotes successive application of zero or more of the productions. Thus, $\mathrm{First}(a)$ contains all symbols with which words can begin that were derived from $a \in \Sigma$. $\mathrm{Follow}(a)$ contains all symbols that can succeed a in a word. These sets are computed by solving a system of inequalities using a fixpoint iteration scheme (see Table 1).

The key observation is that no follow set contains $\mathrm{A_L}$ *and* $\mathrm{A_R}$, $\mathrm{B_L}$ *and* $\mathrm{B_R}$, or $\mathrm{C_L}$ *and* $\mathrm{C_R}$. Thus, the winding does not need to be stored explicitly, except for the first triangle. For all successive triangles it can be inferred.