LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-411859

On-the-Fly Decompression and Rendering of Multiresolution Terrain

P. Lindstrom, J. D. Cohen

April 3, 2009

IEEE Visualization 2009
Atlantic City, NJ, United States
October 10, 2009 through October 16, 2009

Disclaimer

On-the-Fly Decompression and Rendering of Multiresolution Terrain

Peter Lindstrom Jonathan D. Cohen

Lawrence Livermore National Laboratory*

Abstract

We present a streaming geometry compression codec for multiresolution, uniformly-gridded, triangular terrain patches that supports very fast decompression. Our method is based on linear prediction and residual coding for lossless compression of the full-resolution data. As simplified patches on coarser levels in the hierarchy already incur some data loss, we optionally allow further quantization for more lossy compression. The quantization levels are adaptive on a per-patch basis, while still permitting seamless, adaptive tessellations of the terrain. Our geometry compression on such a hierarchy achieves compression ratios of 3:1 to 12:1.

Our scheme is not only suitable for fast decompression on the CPU, but also for parallel decoding on the GPU with peak throughput over 2 billion triangles per second. Each terrain patch is independently decompressed on the fly from a variable-rate bitstream by a GPU geometry program with no branches or conditionals. Thus we can store the geometry compressed on the GPU, reducing storage and bandwidth requirements throughout the system.

In our rendering approach, only compressed bitstreams and the decoded height values in the view-dependent "cut" are explicitly stored on the GPU. Normal vectors are computed in a streaming fashion, and remaining geometry and texture coordinates, as well as mesh connectivity, are shared and re-used for all patches. We demonstrate and evaluate our algorithms on a small prototype system in which all compressed geometry fits in the GPU memory and decompression occurs on the fly every rendering frame without any cache maintenance.

1 Introduction

With the increasing availability of remote sensing technology such as SAR and LIDAR, terrain surfaces on the Earth and neighboring planets are being mapped at astonishing resolution, precision, and scale. As an example, the recent Shuttle Radar Topography Mission (SRTM) mapped about 80% of Earth's land surface at one arcsecond intervals (about 30 meters), resulting in well above one hundred billion data samples. The visualization community has responded well to this increase in size and availability of large data sets, and many algorithms for interactive terrain visualization have been proposed; see, e.g., the recent survey by Pajarola and Gobbetti [25].

As data sets have gotten bigger and CPUs and GPUs become faster and more powerful, with an increasing number of cores, there has not been a proportional increase in the performance of disk and main memory access, causing data transfer to be the main bottleneck in most interactive applications. As a means to combat this trend, data compression coupled with plentiful compute power for doing decompression are being considered as a cost effective approach to boosting effective bandwidth and memory capacity. Consequently, many publications on terrain visualization over the past decade incorporate some form of data compression [2, 3, 6, 9, 10, 12, 21, 29]. Until now, such methods have largely focused on reducing disk space and I/O, with the CPU doing the decompression of the data—often asynchronously using multithreading—and shipping the decoded data to the GPU as needed. Although often effective, without clever prefetching techniques this approach incurs substantial latency before the on-disk compressed data arrives decoded at the GPU, and furthermore consumes precious GPU VRAM, CPU-to-GPU bandwidth, and CPU resources for decoding the data. Ideally decompression would be delayed as long as possible, without ever storing the data decompressed. With the increasing flexibility of today's graphics cards, we show that this is indeed possible by storing the data compressed on the GPU and decoding selected subsets of it on-demand for each rendered frame.

In designing a compressor for terrain data, one must consider the implications of compression. For instance, should the compression be lossless or is some data loss acceptable? The trend has been toward the latter, mostly driven by the need for maximum compression, with the assumption that the source data was acquired with limited accuracy and that the visualization process has to approximate the full resolution data anyway using level of detail techniques. With improvements in scanning technology, accuracy is now approaching the precision available to store the data; nearly without exception as 16-bit integers. Furthermore, a host of GIS applications that combine data analysis and interactive visualization demand that the full-resolution data be accessible without loss. Examples include watershed and flood analysis, military applications like traversability and line-of-sight computation, precise motor grading, climate simulation, and a number of applications that represent height fields other than terrains, e.g. 3D range scans and scalar fields in scientific computing. Remote data servers and data dissemination also benefit from a lossless compressed format that can be efficiently visualized.

In spite of the demand for lossless compression of digital elevation data, surprisingly little work has been done in this area. For these and the reasons above, we focus in this paper primarily on high-speed lossless compression, but allow lossy coding of the lower resolution data in a multiresolution hierarchy without cracks. We represent the terrain as a *4-8 hierarchy* of triangular patches consisting of hundreds of triangles each that supports fine-grained control over the number of triangles per patch. The grouping into patches is done for a number of reasons: to provide efficient batched rendering of graphics primitives, to reduce the mesh adaptation cost, to support view frustum culling, to serve as a unit of paging between different levels of cache, and to impose some level of spatial locality in the compression of the terrain data. These patches can be explicitly managed by a cache manager, with compressed cache levels on external storage, main memory, and the GPU. If desired, an uncompressed cache level may also be maintained on the GPU. In this work, we examine the case where all the compressed data fits on the GPU. In every frame we decode each patch

*Prepared by LLNL under Contract DE-AC52-07NA27344.

to be rendered into a temporary buffer. The decoded height data are combined in a second pass with reusable mesh connectivity as well as geometry and texture coordinates computed on the fly in a vertex program. Triangle normals are optionally computed in a geometry program in this second pass. Hence all that is stored on the GPU is a variable-rate bitstream of encoded height values; meta data such as approximation errors and bounding volumes for mesh adaptation are stored in main memory. This approach makes several novel contributions to the state of the art in fast, multiresolution rendering of compressed terrains:

- **Fast, lossless terrains**: We present a complete terrain codec supporting lossless compression and very fast decompression.
- **Crack-free quantization**: We present an algorithm for lossy compression of coarse-LOD patch vertices with per-patch quantization levels and crack-free display.
- **Optimized linear predictor**: We develop a new, general formulation of linear prediction for raster-scan vertex arrangements requiring fewer mathematical operations.
- **Optimized GPU parallelism**: Our GPU kernel reads variable-bitrate compressed data (most GPU kernels read only fixed-rate data) and contains no branches or conditionals.

Our prototype system demonstrates consistently high decoding throughputs of up to two billion triangles per second and lossless compression rates on the order of 3:1 to 12:1 for large terrains.

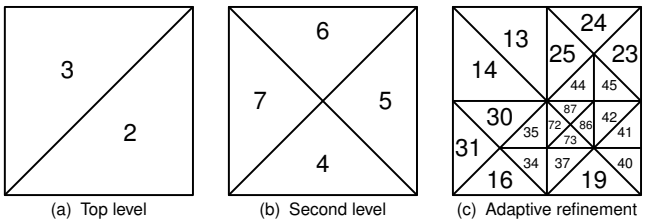
2 Related Work

The terrain visualization literature is vast, and here we focus mainly on work related to multiresolution visualization of regular grids and on approaches to geometry compression. For a more thorough treatment, we refer the interested reader to the recent survey [25].

Early work on adaptive meshing of regular grids [7, 18, 19, 24] was motivated by the need to produce minimal triangulations as quickly as possible on the CPU. As today's GPUs are able to consume individual triangles faster than the CPU can feed them, more recent work has extended this idea from operating on single triangles to larger *patches* of triangles, and differ mostly in whether each patch is uniformly refined [3, 10, 12], adaptively refined [6, 17], or remeshed altogether using irregular connectivity [4]. Like Hwa et al. [12], we rely on regularly-gridded patches as that simplifies decoding and obviates encoding the mesh connectivity.

On the topic of terrain compression, the approaches differ generally in the intended application (interactive visualization versus formats for data distribution), whether the scheme is lossy or lossless, and what is being compressed (e.g. geometry versus connectivity). For non-interactive applications, decoding speed is of secondary concern, thus standard image compression techniques based on wavelet transforms coupled with statistical coding schemes that maximize compression, possibly at the expense of decoding speed, tend to be popular [14, 26]. This approach can also be effective in tile-based interactive methods, where each tile is decompressed on the CPU as it is paged in from disk [2, 3, 10, 15, 21]. In the context of multiresolution rendering, these lossy compression techniques complicate joining the boundaries of different resolution tiles seamlessly. Thus a second crack filling pass may be needed to ensure at least a C^0 continuous surface [6, 21].

Since terrains are predominantly regularly sampled, a large portion of the geometric information need not be stored explicitly but can be derived on the GPU. For example, mesh connectivity and xy (geometry) and uv (texture) coordinates can often be computed on the fly [23, 29], and some methods have explored multiresolution adaptive resampling by essentially treating the height field as a texture [1, 5, 20]. We also take advantage of such implicit information to further reduce memory and bandwidth requirements.



(a) Top level (b) Second level (c) Adaptive refinement

Fig. 1: Adaptive bintree hierarchy of triangle patches. The indexing scheme allows constant time computation of indices for children, parents, siblings, and cousins that form a "diamond," e.g. 13 and 14.

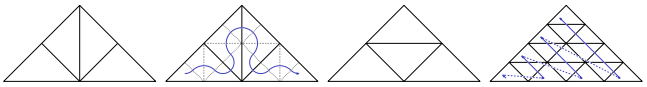

(a) Dual bisection (b) Sierpinski order (c) Quadrisection (d) Raster order

Fig. 2: Ways of refining and traversing triangle patches with (a–b) 4–8 and (c–d) regular connectivity.

A few recent methods have explored decoding compressed terrain data on the GPU. Dick et al. [6] presimplify the terrain via adaptive 4-8 meshing using a progression of object space error tolerances, and then quantize the resulting integer z elevations to 12 bits of precision that, together with 10-bit xy coordinates, are stored as fixed-length codes. The adaptive coarsening results in a mesh with irregular connectivity that in a scheme similar to [9] is then encoded for efficient decompression on the GPU. A number of recent papers address GPU decoding of other types of geometric data, e.g. using (lossy) vector quantization [16, 28] and adaptive [11] and uniform [27] scalar quantization. For lossy compression, one may also consider existing hardware-supported compressed fixed-rate formats for texture maps, e.g. S3TC (aka. DXT), and normal maps, e.g. 3Dc. It is however not immediately clear how these techniques could be retrofitted for efficient, lossless coding.

Though not lossless, the methods in [6, 10, 26], for example, go part of the distance by enforcing bounds on the maximum error introduced. Below we describe a very fast, variable-rate lossless scheme that can exploit lossy compression as part of the multiresolution coarsening process. To our knowledge, this is the first lossless terrain compression scheme that is also amenable to an efficient GPU implementation.

3 Terrain Representation

Before describing our compression scheme we provide a brief overview of the multiresolution representation we use for the terrain. We assume that the input terrain is a uniformly gridded height field with 16-bit unsigned integer elevations that are transformed to world coordinates by a uniform scale factor Δz. The horizontal post spacing is given by Δx and Δy. Like most methods for regularly gridded terrain [7, 18, 19, 24], we rely on a bintree hierarchy formed by 4-8 subdivision (aka. longest edge bisection), but with each bintree node representing a whole patch of triangles, similar e.g. to [4, 12, 17]. Such a hierarchy can be adaptively refined to produce a conforming mesh free of cracks (Fig. 1). Note that the interior of each patch can be arbitrarily tessellated as long as all three patch sides are divided into the same, fixed number of triangle edges. In our case, we restrict ourselves to patches whose interiors are regularly gridded. Unlike [12, 17], who use the same 4-8 refinement within each patch, we use a regular grid connectivity with n "rows" of triangles (Fig. 2), lifting the restriction that n be a power of two for more fine-grained selection of the patch size. Each patch thus has $V = (n+1)(n+2)/2$ vertices and $T = n^2$ triangles, and the terrain dimensions are therefore limited to $(2^m n + 1)^2$ for some non-negative integer m.

For adaptive refinement and crack avoidance, we base our scheme on Lindstrom and Pascucci's SOAR algorithm [19], which stores with each *diamond*—two *cousin* patches that share a hypotenuse—an object-space error term and a bounding sphere that determine when the diamond needs to be refined to meet a screen-space error tolerance. To ensure a crack-free mesh the errors and bounding spheres are inflated where necessary, such that each parent's error/sphere is at least as large as its children's. Furthermore, cousins must agree on a single error and sphere in order to refine together, and thus the nesting relationship is enforced for cousins also (see [19] for details). This information is precomputed once for each patch during encoding and is cached in main memory at run time. We additionally store in memory with each patch the xy coordinates of its first triangle, an offset into the compressed bitstream, and a single quantization parameter, as described in Section 5, for a total of 40 bytes of meta data per patch.

We use a simple binary tree indexing scheme to identify and organize the patches in a breadth-first manner, from coarse to fine resolution. This scheme allows cousin patches to be identified quickly in the offline construction. To generate a view-dependent mesh we make a recursive traversal over the patch hierarchy by indexing into a fixed-size 1D array of patches and accumulate patch IDs for later decompression and rendering. This makes for a particularly simple and efficient implementation.

4 Lossless Compression

We now turn our attention to losslessly compressing the height field over a patch. We assume that height values are uniformly quantized to sixteen bits, and that this is sufficient precision to represent any terrain.[1] As is common in lossless image and mesh coding, we use a linear predictor to estimate each height sample from a small number of already coded samples, and encode the difference with respect to the actual value. In particular, we use parallelogram prediction [13, 30] for the z (height) component; the xy coordinates lie on a regular grid and are thus known. By carefully choosing the traversal of the vertices in a patch, it is possible to use 2D parallelogram prediction only (as opposed to less accurate 1D or constant predictions), and to maximize the number of configurations in which the parallelogram is square (i.e. the triangles form a "diamond"), as that ensures that the vertices involved in the predictor are as close as possible in the domain. Fig. 3 and Fig. 5 illustrate this traversal of the patch, in which diagonal rows of vertices are encoded at a time. The scheme is bootstrapped by storing the three vertices of the first triangle uncompressed.

For the diamond predictor (Fig. 3(c–d)), a sample $z_{i,j}$ on (ascending) vertex row i and (descending) column j is predicted as

$$p_{i,j} = z_{i-1,j} + z_{i,j+1} - z_{i-1,j+1} \tag{1}$$

and a residual $r_{i,j} = z_{i,j} - p_{i,j}$ is computed and encoded. This implies that we have to maintain one row of vertices z_{i-1} in a buffer from which the next row z_i is predicted. Notice that the additive term $z_{i-1,j+1}$ in the previous prediction of $z_{i,j+1}$ is subsequently subtracted off in the expression for $p_{i,j}$. We can thus rewrite $z_{i,j} = p_{i,j} + r_{i,j}$ by expanding the recurrence

$$\begin{aligned} z_{i,j} &= z_{i-1,j} + z_{i,j+1} - z_{i-1,j+1} + r_{i,j} \\ &= z_{i-1,j} + z_{i,j+2} - z_{i-1,j+2} + r_{i,j} + r_{i,j+1} \\ &= z_{i-1,j} + z_{i,i-1} - z_{i-1,i-1} + \sum_{k=j}^{i-2} r_{i,k} \\ &= z_{i-1,j} + s_{i,j} \end{aligned} \tag{2}$$

[1] For higher precision terrains, one could store a per-patch offset into a wider range, as the small full-resolution patches have limited range.

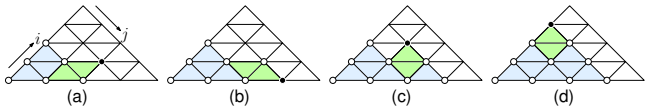


Fig. 3: Linear predictions of one row of vertices (solid) from already decoded vertices (hollow). (a–b) The first two vertices in a row are decoded using parallelogram prediction. (c–d) All subsequent vertices in a row are predicted in a diamond configuration.

Thus, we save one out of every three arithmetic operations by using a residual accumulator s initialized on each row i with $z_{i,i-1} - z_{i-1,i-1}$. Since $z_{i-1,j}$ is no longer needed after it is used to predict $z_{i,j}$, we may drop the row subscript i and overwrite $z_{i-1,j}$ with the next vertex $z_{i,j}$ in the same column j:

$$\begin{aligned} s &\leftarrow s + r_{i,j} \\ z_j &\leftarrow z_j + s \end{aligned} \tag{3}$$

This front-advancing streaming decoder is very efficient and requires buffering only one row of vertices within each patch.

4.1 Residual Coding

As is well-known in the compression literature, the distribution of residuals is often highly peaked around zero, which can be exploited, for example, by entropy coding techniques. Entropy coding is, however, less effective for data whose precision is high with respect to the number of values coded, in part due to difficulties in probability modeling. Furthermore, even Huffman coding requires a nontrivial amount of code to be executed, and may not be well suited for a GPU implementation. Instead, static non-statistical codes that allocate fewer bits to smaller residuals, such as the Elias omega code [8], are a possible alternative. Such variable-length codes must encapsulate both the *value* bits and the bit *length* of each residual, which often incurs a substantial overhead relative to the value bits only (e.g. the omega code requires up to 7 length bits for 16 value bits).

Recently Moffat and Anh [22] proposed the *RBUC code* that exploits coherence by amortizing the length bits over several consecutive residuals. As is done in Elias omega coding, this procedure is applied recursively, implicitly generating a tree representation of the residual stream, so that each level stores the number of bits needed to encode values on the next lower level. The value of each interior node in the tree is thus the maximum $w = \max_i \lceil \log_2(x_i + 1) \rceil$ taken over its child nodes with unsigned values x_i. Since the bottom-level residuals are signed, it is customary to first map them to unsigned ordinals,[2] e.g. in the order $0, -1, +1, -2, +2, \ldots$ RBUC trees are in practice quite shallow: four levels, including the residuals stored in the leaf nodes, suffice to represent residuals as wide as 127 bits, with the root node represented as two bits.

Moffat and Anh's method groups residuals by exhaustively computing the best of several different per-level branching factors in the tree. Our compression scheme is based on RBUC, but uses static branching so that each tree exactly spans the fixed number of residuals $V - 3$ in a patch. To assign branching factors, one could consider combinations of the prime factors of $V-3$. However, for some patch sizes the factors are few and large and lead to poor branching. Instead, we found that a lowest-level branching factor of 4–5 vertices empirically works well for most terrains, with a second-level branching of around 16 nodes. To improve spatial locality, our trees are also constructed so as to avoid grouping residuals from the end of one patch row with those from the beginning of the next. Thus, each row is partitioned into groups of 3–5 vertices, with second-level groups spanning a whole number of rows.

[2] This mapping is used only for computing w in the RBUC tree.

The RBUC nodes are coded in a pre-order traversal of the tree. One key advantage of this is that decompression can be done very efficiently by replacing the pre-order recursion with three nested loops—one for each level of branches in the tree—and by maintaining the "current" bit length on each level. In a GPU implementation, these loops can be unrolled, leading to conditional-free code. Because of this unrolling, it matters little to the decoder that the branching factors vary within each level. Thus, the decoder determines the number of bits to read for the next few nodes, and fetches variable-length residuals from a bit stream. Since residuals are signed, we store them biased by 2^{w-1}, where w is the current residual bit length, to move them into the unsigned range $[0, 2^w-1]$.

In our experience the RBUC scheme yields excellent compression with only minor overhead for representing the bit lengths; usually on the order of 2–3 bits overhead per vertex. In fact, at only 2–4% less compression, RBUC is quite competitive with static (but data-dependent) Huffman codes, in part due to the coherence in residuals resulting from our localized traversal of the vertices. In relation to the Elias gamma code, as used e.g. in [10], RBUC improves compression by 25–60%. RBUC is attractive also in the sense that most read calls return directly usable value bits without the need for table lookups, dependent read calls, or bit by bit parsing of the compressed stream.

5 Lossy Compression

In order to support multiresolution queries, e.g. for view-dependent level of detail, we construct a whole bintree hierarchy of patches, which essentially doubles the patch count. Worse yet, as the resolution decreases, linear prediction performs increasingly worse, such that the majority of the compressed data resides on the upper levels of the hierarchy. One may justifiably question the need to losslessly preserve the coarsened data as the coarsening (subsampling) already incurs data loss. In practice, lossless compression is driven by the need to exactly reconstruct the full resolution data only, and hence it is reasonable to consider lossy compression of coarsened patches.

A simple way of controlling data loss is to simply quantize height values in the upper hierarchy. The approach we take is to first evaluate for each patch the error ϵ_c due to coarsening, and to then allow a proportional quantization error $\epsilon_q = q\epsilon_c$, where q is some fixed constant usually less than one. Since the coarsening and quantization errors may compound, a single unified error must then be recomputed and nested for each patch. That is, we do not use ϵ_c as a hard tolerance that must be met in the quantization process.

Quantization has to be done with care to ensure a crack-free terrain. For example, patch boundaries cannot be quantized arbitrarily. Furthermore, quantizing residuals only, as is done for example in wavelet image coding, is generally not possible, as that would not allow adjacent patches to agree on the reconstructed height values on shared boundaries. Instead, we must quantize the original height values and allow different amounts of quantization of interior and boundary vertices. Furthermore, for an efficient implementation, we restrict the quantization levels to be power of two multiples of the vertical resolution Δz, such that quantization can be implemented using simple bit shifts. The shift is given by

$$k = \begin{cases} \lfloor \log_2 \frac{\epsilon_q}{\Delta z} \rfloor + 1 & \text{if } \epsilon_q \geq \Delta z \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

In selecting the quantization shifts, we first note that all vertices on the finest level must be assigned a zero shift for lossless coding. On the remaining levels, we first classify vertices as boundary and interior (I) vertices. The boundary vertices are further divided into corners (C) and vertices on patch leg (L) and patch hypotenuse (H) edges. Corner vertices may be surrounded by patches at several different resolutions, and hence we simply leave them unquantized. An L vertex in a given patch may also be an L vertex in an adjacent patch on the same level, or it may be an H vertex in a patch one level below. Thus each side of a patch is shared by four patches—two children and their two parents—that must agree on the quantization to use for that side. Since nesting ensures that the children's coarsening errors are no larger than their parents', the precision needed for a vertex is dictated by the finer resolution children. As a result, the shifts for L vertices are governed by other patches, leaving only the shifts for H and I vertices to be determined.

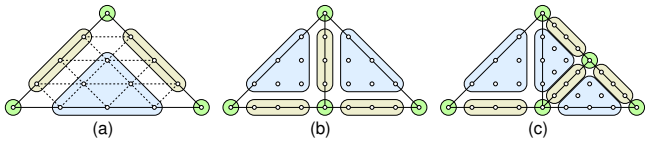


Fig. 4: (a) Classification of patch vertices as corners (green), edges (yellow), and interior/remaining (light blue). A patch leg can be adjacent only to (b) another patch leg at the same resolution or to (c) a hypotenuse of a finer resolution patch. The latter dictates the precision to use for vertices on a shared edge. All corners are stored at full precision.

Here there is no reason to distinguish between H and I vertices, as only the two cousin patches that form a diamond must agree on the quantization to use for H vertices. But these two patches have the same coarsening error, and hence the same shift may be used for the union of their H and I vertices. The resulting classification of vertices and patch neighbor configurations are shown in Fig. 4. Using this classification, three shift values per patch are needed by the decoder: two for the legs and one for the remaining H and I vertices. Only the H/I shift is stored with a patch; the two L shifts are the H/I shifts of its children.

Note that quantization has an influence on prediction. In particular, all vertices involved in a prediction must be reduced to the precision of the vertex being predicted to avoid introducing unwanted low-order bits on patch boundaries. We thus shift out such low-order bits where necessary, perform the prediction at a unified precision, add in the low-precision residual, and then left shift the result as needed. In practice, this step is necessary only for the C and L vertices in the first triangle of a patch due to the regular connectivity of patches and our traversal order from diamond interior to boundaries, i.e. from low to high precision vertices.

6 GPU Decompression

The encoder described above lends itself to a straightforward decoder implementation on the CPU that takes a variable-length bitstream for a patch as input and outputs a sequence of integer height values, which can be rendered using indexed triangle lists. We now describe how to perform the same task efficiently on the GPU.

Current-generation graphics cards support not only programmable shaders for vertices and fragments, but can even generate novel connectivity and geometry in so-called *geometry programs*. Following the execution of the vertex program on its vertices, each rendering primitive becomes a thread of execution that can emit a long list of points, triangles, or triangles strips that get sent directly to the rasterization stage. In fact, using parallelogram prediction one could easily generate both xy and z coordinates using vector instead of scalar arithmetic in the predictor (with no residual term for xy), and output one triangle strip per row in the order shown in Fig. 2(d). Texture coordinates could similarly be generated, and normals computed on the fly. All these coordinates could conceptually be streamed directly from registers to the rasterizer without ever being written to memory.

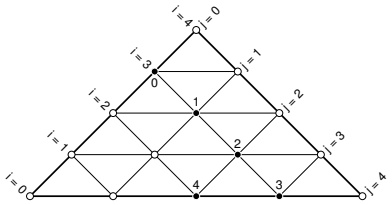


Fig. 5: Numbering of vertices on the advancing front just after row i $= 3$ has been decoded. Vertex 4 is needed to predict row i $= 4$.

```
for (int i = 2; i <= n; i++) {                  // for each row...
  short s = z[i-2] - z[i] + residual();         // initialize accumulator
  z[i+1] = z[i-1];                              // keep first in previous row
  z[i-1] += s;                                  // second vertex in this row
  z[i] = z[i-1] + z[i+1] - z[i-2] + residual(); // first vertex in this row
  emit(z[i]);                                   // output first vertex
  emit(z[i-1]);                                 // output second vertex
  for (int j = i - 2; j >= 0; j--) {            // for each column...
    s += residual();                            // update accumulator
    z[j] += s;                                  // update vertex
    emit(z[j]);                                 // output vertex
  }
}
```

Listing 1: Decoder pseudocode. The loops are in actuality unrolled and the `residual()` call, which may involve multiple reads, is inlined.

```
uint read(inout STREAM s, uint n) {       // read n bits from bitstream s
  s.bits -= n;                            // subtract number of bits buffered
  uint m = s.bits & 16;                   // number of bits to fetch (0 or 16)
  s.bits += m;                            // add number of bits fetched
  s.word <<= m;                           // make room for more bits
  m >>= 4;                                // number of words to advance
  s.word += texBUF(s.tex, s.ptr).x & -m; // fetch bits and insert or discard
  s.ptr.x += m;                           // advance pointer
  uint value = s.word >> s.bits;          // extract value bits
  s.word -= value << s.bits;              // remove from register
  return value;                           // return n bits
}
```

Listing 2: Variable-length reads from bitstream.

Unfortunately the semantics of OpenGL and hardware limitations place restrictions on the amount of data that a geometry program can output: 1,024 scalar attributes on the NVIDIA GeForce GTX 280 GPU. Emitting all the necessary attributes for rendering a complete patch would severely limit the patch size and therefore compression, due to the need to replicate vertices on patch boundaries. However, the geometry shader is still the most promising stage in the graphics pipeline for decoding patches. The 4,096 bytes of output available to a geometry program is significantly more than the 128 bytes available to a fragment program.

Fortunately an option exists to bypass the rasterizer by writing the output to a buffer for later access during a second rendering pass. We employ this approach of emitting and temporarily buffering only height values, as it also better exploits parallelism and improves decoding throughput. If output size was the only constraint, we could emit patches with as many as 62 rows (our unrolled program instruction count currently limits us to 35 rows). Note that only a buffer large enough to hold the actual z values of the displayed geometry (after view frustum culling) is needed, or on the order of a few megabytes. Although the result could be cached and reused across frames, the penalty for not doing so is quite small, and the lack of caching simplifies the implementation.

Our GPU decoder is implemented in the Cg high-level language and compiled to OpenGL shader assembly code (we find that examining the assembly code helps guide our optimization). The decoder takes as input a pointer to a *texture buffer* (essentially a contiguous chunk of GPU memory), an offset into the buffer to the beginning of the patch's compressed bitstream, and three quantization shifts (see Section 5). The first three height values are read verbatim from the bitstream and are output. We then proceed to decode vertices one row at a time. As in the case of encoding, only one row of height values is needed to generate the next row, and can be stored in registers. For each vertex, we make a prediction, read the next residual from the bitstream and left shift it according to the vertex's quantization level, and finally emit the reconstructed height value. These steps are summarized by the pseudocode in Listing 1, which shows the two nested loops over the vertices that we manually unroll using a procedural code generator to avoid conditionals and maximize the opportunity for SIMD parallel execution.

The most challenging aspect of the decoder implementation is making the variable-length reads from the bitstream efficient, which is of particular importance since the RBUC scheme requires more reads than there are vertices in a patch. We implement this in a conditional free manner by making redundant fetches from a texture buffer via a pointer that is advanced only when necessary. Bits from the bitstream enter a 32-bit register from the right 16 bits at a time (or are discarded if already fetched), and at most 16 bits are read from the left of this register at a time. Underflow on the number of buffered bits in the register signal that another 16 bits are needed, and causes the pointer to be advanced. The resulting Cg code (Listing 2) compiles to 11 assembly instructions per read. In all, we require on average 22.2 instructions per vertex without and 23.5 instructions with quantization. As already mentioned, the entire decoder is thus free of conditionals and loops, and executes a fixed number of instructions for each patch.

7 Rendering

We have integrated the geometry decoder into a three-stage *adapt–decode–draw* terrain rendering prototype system. In this prototype, the compressed bitstreams associated with all patches are stored on the GPU, along with almost all the data necessary to decode and draw the patches. Every frame, the cut is adapted to the current error threshold and viewing parameters, the bitstreams of all nodes on the cut are decoded in parallel into a small buffer on the GPU, and the resulting decompressed heights are used to draw the patches as triangles. We do not maintain a cache of decoded geometry in our prototype, although that would clearly be feasible and useful for some rendering systems.

The adapt phase uses a screen-space error tolerance (in pixels) and a set of viewing parameters to produce a *cut* of nodes that are just beneath the error tolerance and not culled by the viewing frustum. A top-down traversal of the bintree patch hierarchy is performed on the CPU to produce this cut, which is organized into bins to minimize graphics state changes in the draw phase.

In the decode phase, the system issues a sequence of patch decode commands in the form of OpenGL point primitives. Each point primitive has the offset of a patch into a texture buffer containing the bitstream, as well as the three required shift parameters for maintaining proper quantization, packed as raw bits into two floating point variables. Each point primitive is passed through a trivial vertex program, which just unpacks the four values before handing them to the main decoder running as a geometry program. A *transform feedback buffer* captures the geometry program outputs for use in the draw phase. The rasterizer is set to discard fragments in this phase, so no fragment program is bound or executed.

Finally, the draw phase renders the decoded patches as indexed OpenGL triangle set primitives. A single set of vertex indices stored on the GPU is reused for every patch to issue the proper vertices for rendering. A vertex program binds a height value from the decoded height buffer with canonical xy-coordinates from a properly-oriented patch, applying patch translation, LOD scaling, and the modelview-projection matrix to produce $xyzw$-coordinates in clip space, xyz-coordinates in object space for normal computation and 1D texture mapping, and uv-coordinates for 2D texture mapping.

If a flat shading rendering mode is enabled, a flat shading program is applied in the geometry shader during the draw phase. This program can access all three triangle vertices to produce a face normal, which we use in our prototype to perform a simple dot product

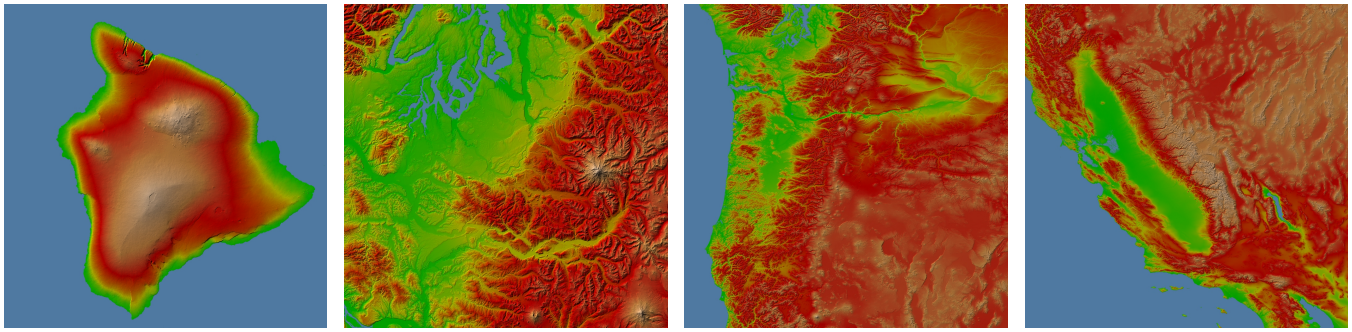


Fig. 6: Shaded relief maps of the data sets we use. From left to right: Hawaii, Puget Sound, Pacific NW, and CalNev.

Data set	Dimensions	$\Delta x = \Delta y$	Δz	Range	# Rows	# Patches	Size (MB) : Compression Ratio			
							Original	$q = 0\%$	$q = 25\%$	$q = 50\%$
Hawaii	$16,385^2$	10 m	1.0 ft	13,781	16	4 M	512	115 : 10.7	107 : 11.4	103 : 11.9
Puget Sound	$16,385^2$	10 m	0.1 m	43,930	16	4 M	512	379 : 3.2	302 : 4.1	289 : 4.2
Pacific NW	$24,577^2$	$1'' \simeq 30$ m	1.0 m	4,384	24	4 M	1,152	555 : 4.7	477 : 5.4	440 : 5.9
CalNev	$32,769^2$	$1'' \simeq 30$ m	1.0 m	4,412	16	16 M	2,048	1,004 : 4.8	896 : 5.5	832 : 5.9

Table 1: Compression results for four large terrains. The two largest terrains are represented in geographic coordinates at one arcsecond ($1''$) resolution. *Range* refers to the maximum integer elevation value (zero being the minimum in all cases), and is thus an indicator of the intrinsic precision of the data. The compression ratio relates the size of the full hierarchy with and without (16 bits/vertex) compression and quantization q (mb/mv in the terminology of Fig. 7).

for diffuse illumination. Otherwise, no geometry program is bound, and the vertex program results go directly to the rasterizer. In either case, one of a number of fragment programs is bound to produce effects such as flat shading with a 1D color ramp (indexed by height value), 2D texturing, contour line extraction, etc.

It is worth making a few key observations here about optimizing these phases. Although the adapt phase is quite simple and efficient at run-time, a full-scale system would pipeline this phase on the CPU with the decode and draw phases of the previous frame. Also, for small patch sizes, it may be useful to adapt at a coarser granularity than the decoding and drawing to avoid a CPU bottleneck.

The decoding program is highly optimized, but its overall performance also depends on a number of factors besides instruction count. As we show in Section 8, the number of output bytes per decode thread has a big impact on performance due to limited shared memory for the on-chip output buffer (before writing to the off-chip transform feedback buffer). The NVIDIA GT200 series card we are using increased the size of the output storage by a factor of six over the GeForce 9 series to reduce this bottleneck. Because the `emit` commands issue four-byte attributes, we pack two height values into each `emit`to fully utilize this precious space. Also, the use of point primitives to issue patches (as opposed to, say, triangle primitives) keeps the number of vertex program threads in the decode phase to a minimum, allowing more GPU multi-processors to be tasked as geometry program threads for the actual decoding.

To optimize the draw phase, the cut is organized by bitstream buffer number (the largest terrains require splitting the stream into a few texture buffers), orientation (patches have one of 16 canonical orientations), and hierarchy level. Only a height buffer offset and translation are set before issuing each patch. Also, we found issuing a single `glDrawElements` call per patch using triangle sets to be significantly faster than using `glMultiDrawElements` containing multiple small triangle strips per patch.

8 Results

We evaluate our compression and decoding methods on a PC running Fedora Core 10 with 3 GHz Dual Xeon CPUs and an NVIDIA GTX 280 GPU with 1 GB of VRAM. In our study we use four data sets (see Fig. 6 and Table 1): a slightly cropped and padded (to make it square) version of Hawaii's Big Island available at `http://duff.geology.washington.edu/data/raster/tenmeter/hawaii/`; the Puget Sound benchmark data set from Georgia Tech; an area of the Pacific Northwest covering roughly 41–48N latitude, 118–125W longitude; and the California and Nevada states, covering the nearly 10×10 degree area 32.5–42N, 114–124W. The latter two SRTM data sets are available at `http://www2.jpl.nasa.gov/srtm/`. We performed simple hole filling to patch up small voids in these two data sets.

8.1 Compression

We first investigate how well these data sets compress using our method. As patches overlap on their boundaries and because each patch incurs some fixed overhead, there's a compression penalty associated with choosing patches to be too small. On the other hand, choosing very large patches may ultimately lead to poor spatial locality in the row-by-row traversals we make. Furthermore, as we shall see, large patches decompress slower. Fig. 7 highlights the trade-off between patch size and compression rate for Hawaii and Puget Sound. We plot the storage cost for the whole hierarchy (mb) as well as just the bottom-most, full-resolution level (sb) averaged over all vertices in the hierarchy (mv) and just the vertices in the original, single-resolution data set (sv). In our prototype any patch size that is not a power of two requires cropping the terrain (to as little as 28% of the total area for 17 rows), which is the cause for the kinks in the curves. From the top two curves it is clear that the upper levels, which represent wider spacing between samples, compress worse than the bottom level as the per vertex cost everywhere more than doubles between these curves. This further motivates the need for quantization on those upper levels.

Because the hierarchy we build in fact adds functionality in terms of multiresolution queries without requiring access to the entire full-resolution data, we believe it is appropriate to also compare the per-vertex storage cost in the hierarchy with and without compression. The green curves show that a patch size of 16 and above provides a fair trade-off in compression, and the per-vertex cost for Hawaii approaches 1.2 bits/vertex. Admittedly this data set contains a large amount of water at zero elevation, which improves compression. In practice, however, it is common for high-

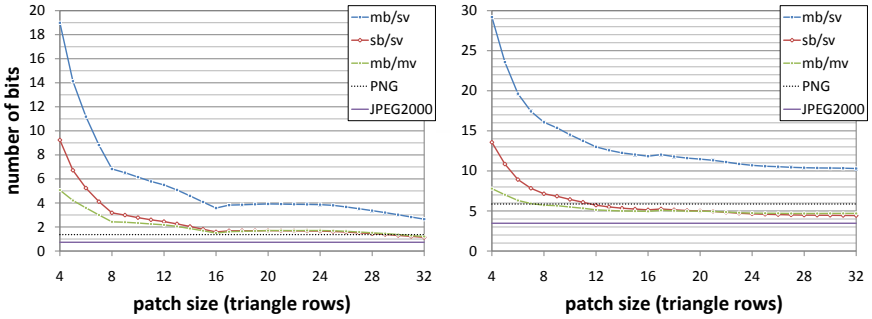


Fig. 7: Lossless compression as a function of patch size for the Hawaii (left) and Puget Sound (right) terrain. *mb* refers to the total number of bits for the multiresolution terrain; *sb* the number of bits for the patches in the single, finest resolution only. Similarly, *mv* is the total number of vertices stored in all patches of the hierarchy, and *sv* the number of vertices in the single-resolution input terrain.

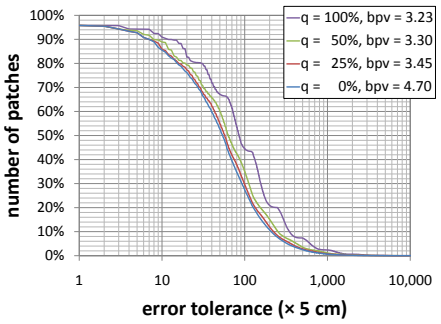


Fig. 8: Effect of quantization q and object-space error tolerance on Puget Sound patch count. The 1.25 bits per vertex reduction in storage incurs only a modest increase in patch count at $q = 25\%$.

resolution data sets to contain large irregularly shaped voids in regions where no data was acquired (either for legal or practical reasons), and without compression such regions would have to be represented with as many bits as the actual data. In actual use cases over land we have seen as much as 30:1 lossless compression of regular grids with partially missing data.

As expected, Puget Sound does not compress as well, but we achieve a respectable 4.95 bits/vertex (mb/mv) at 16 rows and 5.14 bits/vertex when considering only the full resolution (sb/sv). This compares favorably with the lossless PNG image format (5.87 sb/sv), which relies on zlib as a much more complex back end coder. At 32 rows our method is about one bit per vertex less efficient than lossless JPEG2000, though in fairness the complexity of the JPEG2000 format makes it quite ill suited for fast decoding.

Fig. 8 illustrates the benefit of quantizing upper levels. As quantization often increases the measured error over a patch, we would expect a net increase in patches (and thus rendered triangles) when refining the mesh to a given error tolerance as compared to using no quantization. For these curves we refined the mesh to a given object-space error tolerance and counted the corresponding number of patches. It is evident from this graph that at low quantization levels, e.g. $q = 25\%$, the increase in patches is only a few percent. The improvement in compression is, however, quite significant; a reduction from 4.70 to 3.45 bits/vertex. Finally, we note that although the vertices on upper levels are simply subsampled from finer resolutions, the use of quantization could also be coupled with higher-quality low-pass filtering with only minor code changes.

A comparison with other terrain compressors is not straightforward, as few support lossless compression, whereas in the lossy case it is not obvious how to evaluate rate distortion over multiresolution hierarchies. We suspect that methods designed solely for lossy coding are, however, likely to improve on our results.

8.2 Decoding and Rendering

We now consider the performance of the GPU-based decoder taken alone and in the context of interactive rendering. Fig. 9 shows how the decode throughput varies with the number of decoded triangles. Two aspects of the performance characteristics of the decoder are quite clear from the graph. First, throughput increases with input size. This is not surprising, because the available parallelism (number of parallel execution threads) increases with the problem size, and some overhead may be increasingly amortized as well. Second, we see three widely-spaced performance tiers for different numbers of rows-per-patch. These performance tiers are determined by the number of bytes produced per patch (and thus per thread). In general, within a tier, performance increases with the number of rows. Some sub-tiers are visible as well, due to other discrete constraints on allocation of GPU resources. Note that the curves increase in

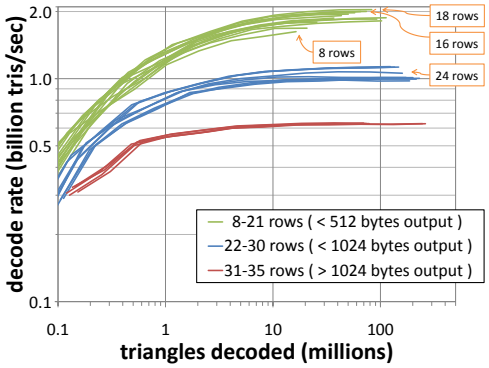


Fig. 9: Decode throughput for different patch sizes. Evidently the number of bytes output by the geometry program to a GPU memory buffer influences the amount of parallelism possible, resulting in three distinct tiers.

length with increasing patch size due to the patch-level granularity of our measurements, but this fortuitously makes the graph somewhat easier to interpret.

We also collected performance data of the overall rendering system prototype by flying over each terrain in a circular path with a number of different error thresholds and rendering settings. Fig. 10 plots the performance for the CalNev and Puget Sound data sets, each with a screen-space error threshold of half a pixel. For each frame, the filled area plots the number of triangles on the cut (note that this is an error-threshold system, so we are not currently targeting a particular triangle count or frame rate). The green curve shows the decoder throughput. Notice that at between 13 and 15 million triangles on the cut, we achieve over 1.9 billion triangles per second, or roughly 95% of our peak throughput. The decoding throughput is more sensitive to variation in triangle count than the draw throughput because parallelization occurs at a per-patch rather than per-triangle granularity. The purple curve plots the adapt throughput, which is reasonably high and constant at this error threshold. The dark blue curve plots the drawing throughput for 2D texture mapped rendering with no normal computation or real-time illumination, and the light blue curve shows the total system throughput (all three stages) when rendering in this mode. Similarly, the dark red and pink curves show the drawing and total throughput when rendering with normal computation and diffuse shading with a 1D color ramp lookup. The reduction in throughput in this mode is due to binding the geometry shader for normal computation. Interestingly, the performance decrease is the same even if we do no actual computation in that geometry shader; it is due solely to retasking multi-processors to geometry shading as opposed to vertex and fragment shading. Overall, we see that decoding throughput ranges roughly from 3–4 times the drawing throughput (in 2D texturing mode).

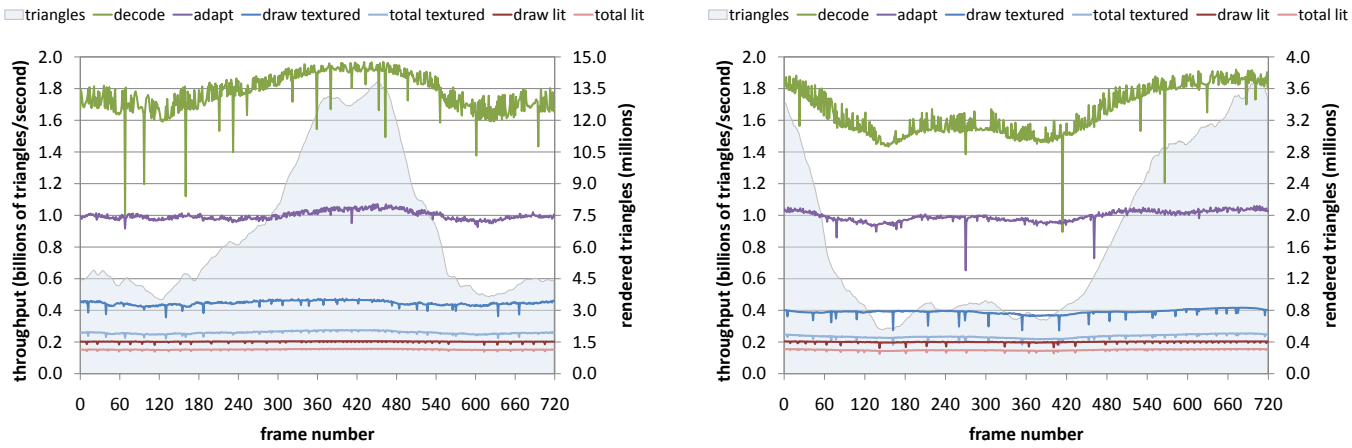


Fig. 10: View-dependent triangle count and throughput for the various stages of our pipeline over 720-frame fly-throughs of the 32769^2 CalNev (left) and 16385^2 Puget Sound (right) data set. The error tolerance was 0.5 pixels. We show in blue the throughput for drawing 2D textured terrain without lighting, and in red lit terrain with on-the-fly normal computation and 1D texturing. The decode and adapt throughputs are insensitive to rendering mode. The occasional downward spikes are due to timer inaccuracies caused by involuntary context switching.

9 Conclusion

We have presented a fast, lossless compression codec for terrains on the GPU, and demonstrated its use for interactive visualization. We build a hierarchical representation that combines streaming, lossless compression at the finest resolution with seamless, lossy quantization at coarser mesh resolutions. Our compression algorithm achieves compression rates of 3:1 to 12:1 over such a hierarchy without compression. As larger and larger data sets become available, such compressed representations become increasingly useful, providing greater flexibility in when and where data decompression takes place in a processing pipeline. In our prototype system, decoding rates are at least 3–4 times the drawing rates, with a peak decoding throughput over 2 billion triangles per second, making it quite acceptable to decode all rendered triangles every frame.

There are several avenues for future work in this area. Building directly on this codec, we envision encorporating this work into a larger-scale rendering system, with data caches on the CPU and GPU managing memory locality of massive, out-of-core data. As future algorithmic research, it will also be interesting to expand our method to handle more general mesh topologies and multi-channel geometry images, all with decompression happening on the GPU.

References

[1] A. Asirvatham and H. Hoppe. Terrain Rendering Using GPU-Based Geometry Clipmaps. *GPU Gems 2*, chap. 2. Addison-Wedley Professional, 2005.

[2] F. Bettio, E. Gobbetti, F. Marton, and G. Pintore. High-quality networked terrain rendering from compressed bitstreams. *Web3D*, 37–44. 2007.

[3] J. Bösch, P. Goswami, and R. Pajarola. RASTeR: Simple and Efficient Terrain Redering on the GPU. *Eurographics Areas Papers*, 35–42. 2009.

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

[5] C. Dachsbacher and M. Stamminger. Rendering procedural terrain by geometry image warping. *Eurographics Symposium on Rendering*, 103–110. 2004.

[6] C. Dick, J. Schneider, and R. Westermann. Efficient Geometry Compression for GPU-based Decoding in Realtime Terrain Rendering. *Computer Graphics Forum*, 28(1):67–83, 2009.

[7] M. A. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. *IEEE Visualization*, 81–88. 1997.

[8] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[9] T. Gerstner. Multiresolution visualization and compression of global topographic data. *Geoinformatica*, 7(1):7–32, 2003.

[10] E. Gobbetti, F. Marton, P. Cignoni, M. D. Benedetto, and F. Ganovelli. C-BDAM—Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering. *Computer Graphics Forum*, 25(3):333–342, 2006.

[11] X. Hao and A. Varshney. Variable-precision Rendering. *Symposium on Interactive 3D Graphics*, 149–158. 2001.

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

[13] M. Isenburg and P. Alliez. Compressing polygon mesh geometry with parallelogram prediction. *IEEE Visualization*, 141–146. 2002.

[14] D. B. Kidner and D. H. Smith. Advances in the data compression of digital elevation models. *Computers and Geosciences*, 29(8):985–1002, 2003.

[15] J. K. Kim and J. B. Ra. A real-time terrain visualization algorithm using wavelet-based compression. *Visual Computer*, 20(2–3):67–85, 2004.

[16] S. Lefebvre and H. Hoppe. Compressed random-access trees for spatially coherent data. *Eurographics Symposium on Rendering*, 339–349. 2007.

[17] J. Levenberg. Fast View-Dependent Level-of-Detail Rendering Using Cached Geometry. *IEEE Visualization*, 259–266. 2002.

[18] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. Turner. Real-Time, Continuous Level of Detail Rendering of Height Fields. *ACM SIGGRAPH*, 109–118. 1996.

[19] P. Lindstrom and V. Pascucci. Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):239–254, 2002.

[20] Y. Livny, N. Sokolovsky, T. Grinshpoun, and J. El-Sana. A GPU persistent grid mapping for terrain rendering. *The Visual Computer*, 24(2):139–153, 2008.

[21] F. Losasso and H. Hoppe. Geometry clipmaps: Terrain rendering using nested regular grids. *ACM Transactions on Graphics*, 23(3):769–776, 2004.

[22] A. Moffat and V. N. Anh. Binary codes for non-uniform sources. *Data Compression Conference*, 133–142. 2005.

[23] K. Niski, B. Purnomo, and J. D. Cohen. Multi-grained Level of Detail Using a Hierarchical Seamless Texture Atlas. *ACM Symposium on Interative 3D Graphics and Games*, 153–160. 2007.

[24] R. Pajarola. Large Scale Terrain Visualization Using The Restricted Quadtree Triangulation. *IEEE Visualization*, 19–26. 1998.

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

[26] R. V. Panchagnula and W. A. Pearlman. Near-lossless compression of digital terrain elevation data. *Proceedings of the SPIE*, vol. 5308, 331–342. 2004.

[27] B. Purnomo, J. Bilodeau, J. D. Cohen, and S. Kumar. Hardware-Compatible Vertex Compression Using Quantization and Simplification. *Symposium on Graphics Hardware*, 53–61 and 117. 2005.

[28] J. Schneider and R. Westermann. Compression Domain Volume Rendering. *IEEE Visualization*, 293–300. 2003.

[29] J. Schneider and R. Westermann. GPU-friendly high-quality terrain rendering. *Journal of WSGC*, 14(1–3):49–56, 2006.

[30] C. Touma and C. Gotsman. Triangle mesh compression. *Graphics Interface*, 26–34. 1998.