Decoding of Large Terrains Using a Hardware Rendering Pipeline [†]

James E. Fowler[1,3], **John van der Zwaag**[3], **Shivaraj Tenginakai**[2,3],

Raghu Machiraju[2,3], **Robert J. Moorhead**[1,3]

[1]*Department of Electrical and Computer Engineering*
[2]*Department of Computer Science*
[3]*NSF Engineering Research Center for Computational Field Simulation*
Mississippi State University, Mississippi State, MS

Abstract

In this paper, we present a simple approach to the quantization of vertex-coordinate values of a very large terrain dataset, allowing both efficient representation and timely rendering of the data. We focus on uniform scalar quantization, which, although theoretically suboptimal in general, is shown to produce the same level of accuracy as a Lloyd-Max optimal quantizer for the terrain data under consideration. We find that a rate of quantization on the order of 6 bits per vertex results in visually lossless rendering of a very large terrain, allowing fly-throughs of high quality at interactive frame rates. Our novel approach to uniform scalar quantizer represents the inverse-quantization operation as a homogeneous matrix, allowing the rendering of the terrain data directly from the encoded representation with inverse quantization taking place automatically in the hardware rendering pipeline without the assumption of any special decompression engine. Additionally, our formulation is general and therefore amenable to incorporation in more sophisticated geometric-compression systems.

1. Introduction

Modern applications of computer graphics call for the processing of geometric models of a complexity that is increasing faster than graphics systems are evolving. The challenges posed by these complex datasets include not only the timely rendering of highly detailed geometric models, but also the transmission and storage of enormous quantities of data [**?**]. For example, the application of interest considered in this paper is the rendering of very large terrains such as those arising in many geographic information systems (GIS). The data input to such a terrain-rendering system usually takes the form of a digital elevation map (DEM) providing a 3D terrain surface; coupled to this 3D surface is an image mosaic providing terrain texture. Typically, the DEM data is delivered as a floating-point array of size on the order of $1,200 \times 1,200$ vertices and may give rise to 1.4 million or more polygons when triangulations between grid points of the DEM are generated. With larger DEMs, the sheer number of polygons can overwhelm many terrain-rendering systems. Consequently, the DEM data is often broken into tiles, and some form of level-of-detail (LOD) hierarchy is employed (e.g., a quad-tree [**?, ?**] or progressive mesh [**?**]). Although the use of tiling and LOD techniques certainly help make the data more manageable, the sheer size of the terrain datasets can still tax the memory subsystems of even the most well-endowed machines. Often one resorts to caching of tiles to make more efficient use of available memory; however, this recourse usually results in sluggish performance. In order to achieve desired rendering speeds with limited memory resources, modern graphics systems are increasingly employing *geometric compression*.

In general terms, geometric compression involves the reduction of the size of geometric datasets through the simplification and efficient representation of geometry. Clearly geometric compression yields reduced

---

storage space for general computer-graphics applications as well as more efficient use of transmission bandwidth for remote visualization systems; in addition, if one can render directly from the compressed dataset, significant speedup of rendering operations is possible. Although the simplification of geometric data structures has been considered in various literature for some time, the concept of geometric compression, which couples *simplification* with efficient *encoding* mechanisms, is generally considered to have originated in the landmark work by Deering [?]. Among others, Taubin and Rossignac [?] suggested that a geometric-compression system be divided conceptually into three types of processes: polyhedral simplification, connectivity encoding, and geometry encoding.

The process of *polyhedral simplification* is aimed at reducing the number of vertices, and, consequently, the number of polygons, in a geometric dataset [?]. Polyhedral-simplification techniques necessarily alter the connectivity of the dataset; additionally, the positions of the vertices that survive simplification may possibly be different from those in the original dataset. Heckbert and Garland present a survey of a number of proposed methods in [?]. *Connectivity encoding* refers to the process of removing redundancy in the connectivity of the dataset [?]. One of the most common examples of connectivity encoding is that of triangle strips, a method of representing a triangle mesh involving the implicit reuse of two previous vertices with the current vertex to define the current triangle. Other, more sophisticated approaches to connectivity encoding have been proposed by Deering [?] and Taubin and Rossignac [?]. The final component of geometric compression is the most fundamental: *geometry encoding* consists of producing an efficient representation for the geometric information stored at each vertex. Such information may include the 3D position coordinates of the vertex, the normal vector at the vertex, and the texture coordinates of the vertex [?]. Although a geometric-compression algorithm may or may not incorporate polyhedral simplification and connectivity encoding, any geometric model amenable to computer-based processing must at least involve some degree of geometry encoding.

Specifically, vertex information, such as xyz coordinates and normal vectors, models continuous-value quantities from the real world which must be discretized through some form of quantization to be used in a computer. Often this quantization is implicit in that finite-precision floating-point number representations are used; however, explicit quantization strategies permit more compact data representations and greater control of the accuracy with which the data is represented. Optionally, additional data-compression methods of greater sophistication may be employed following quantization to further shrink the size of the encoding. For example, predictive techniques have been proposed [?, ?] to exploit the smaller variance and dynamic range of residual values for greater coding efficiency, while lossless data-compression techniques such as entropy coding (e.g., Huffman coding [?] or arithmetic coding [?]) may be employed to remove statistical redundancy.

A main requirement in our application is that the terrain renderer be able to handle extremely large terrain datasets. Although some kind of paging algorithm could be used to access large datasets, it is more desirable that the rendering system occupy only a small footprint in memory despite the amount of data it might be rendering. Such restricted memory use would facilitate rendering at interactive frame rates; additionally, files required by the system would be smaller and more easily transportable, reducing system overhead for data input and enabling remote visualization applications. To achieve these goals, geometric compression is proposed for use not only in the files stored on the file system, but also in the data stored in memory during rendering. However, in order to ensure consistent and interactive frame rates, the system can not require a large amount of overhead to decompress the data into a usable form for rendering. Consequently, the focus of this paper is the development of a very simple approach to geometric compression that is easily inverted and allows rendering directly from the encoded data.

Specifically, we consider the application of quantization to the vertex coordinate values to produce a compact representation of a terrain model. We restrict our attention to quantization since it is fundamental and necessary to any representation of geometry and because, as we show in what follows, it is easily incorporated into modern graphics-rendering systems without the addition of *any* computational overhead.

The advantages of our approach lie in that 1) the compressed dataset is resident in memory and is passed *directly* in compressed form to the rendering subsystem, 2) decompression is performed automatically in the graphics-rendering pipeline in hardware *without* requiring any specialized geometric-decompression engine, 3) the performance of our approach is very close to that of the optimal quantizer for the dataset, and 4) the formulation is general and therefore amenable to incorporation in more sophisticated geometric-compression systems. We note that the approach to quantization we present here may be used easily in conjunction with any of the number of polyhedral-simplification algorithms discussed in [**?**]; on the other hand, connectivity-encoding techniques (e.g., [**?, ?**]) can also be used, but will likely require additional computational overhead or specialized hardware to decode the connectivity.

In the following, we outline our approach to quantization for geometric compression and illustrate how we have incorporated this approach into a terrain-rendering system. In the next section, we review the theory behind scalar quantization. We follow with Section 3 wherein we overview past use of quantization in geometric models and then present our approach to the same. In Section 4, we describe our terrain-rendering application and the details behind the incorporation of our proposed approach to quantization within the terrain-rendering system. Results from our terrain-rendering activities are presented in Section 5, which we follow with some concluding remarks.

2.   Scalar Quantization

Since real-world quantities are continuous value (analog) while modern computers deal exclusively with discrete values (digital numbers), the processes of analog-to-digital and digital-to-analog conversion are fundamental to any digital system. The general process of converting continuous-valued quantities to discrete values is known as *quantization* [**?**], while the operators that perform such quantization are known as *quantizers*. Quantizers may be scalar (one-dimensional) or vector (multidimensional) depending on the dimensionality of the input. Although vector quantizers are theoretically more efficient, scalar quantizers are often preferred in practice due to their simplicity.

Mathematically, a scalar quantizer is a mapping from the real numbers $\Re$ to a set $I \subseteq \mathcal{Z}$ of integer indices,

$$Q : \Re \to I.$$

The quantization of real number $x \in \Re$ is integer index i,

$$Q(x) = i \in I.$$

The corresponding *inverse quantizer* produces an approximation $\hat{x} \in \Re$ to original value x given the index i,

$$\hat{x} = Q^{-1}(i) = Q^{-1}(Q(x)).$$

In general, $\hat{x}$ will not be equal to x, so the quantizer introduces some distortion; this distortion is measured with metric d_Q,

$$d_Q : \Re \times \Re \to \Re.$$

The most commonly used metric is the squared error,

$$d_Q(x, \hat{x}) = (x - \hat{x})^2.$$

Assuming that the dynamic range of x is limited to the interval $[x_{\min}, x_{\max}]$ on the real-number line, the usual implementation of a general scalar quantizer takes the form of a lookup table relating disjoint subintervals in $[x_{\min}, x_{\max}]$ to integer indices. In this case, set I is finite subset of $\mathcal{Z}$, and N,

$$N = |I|,$$

3

gives the number of quantizer intervals; Q is often referred to as an N-level quantizer. The corresponding inverse quantizer consists of another lookup table that relates each integer index to a certain reproduction value $\hat{x}$. We note that scalar quantization is an operation that is both *lossy* and *nonlinear*. Below we overview two important classes of scalar quantizers: *optimal* and *uniform*.

2.1 Optimal Scalar Quantization

The field of information theory (see, for example, [**?**]) defines an *optimal* scalar quantizer as a quantizer that produces the minimum distortion for x for a fixed number of levels N. These optimal quantizers must be tailored to the probability density function of the input quantity x, and the distortion measured as a probabilistic expectation over this density function. Specifically, if the probability density of x is $p(x)$, then an optimal quantizer Q^* is

$$Q^* = \arg\min_Q E\left[d_Q(x, Q(x))\right] \tag{1}$$

where the minimization is over all possible quantizers Q, and the *average distortion* is

$$E\left[d_Q(x, Q(x))\right] = \int p(x)\, d_Q(x, Q(x))\, dx. \tag{2}$$

In practice, the well known Lloyd-Max algorithm (see [**?**]) is capable of designing optimal quantizers for x using a training set of data. The Lloyd-Max algorithm iteratively improves upon a given initial quantizer, adjusting the size of quantizer intervals and the positions of reproduction values until convergence to the optimal quantizer. In general, this optimal quantizer has reproduction levels clustered in regions of relatively high probability, while in regions of low probability, reproduction levels are less densely located. As a result of this probability-based clustering, highly probable values of x are quantized with less distortion, resulting in an overall minimal expected distortion.

2.2 Uniform Scalar Quantization

Because of its iterative nature, the design of optimal quantizers using the Lloyd-Max algorithm can be computationally expensive. Additionally, implementations of the quantizer as well as the inverse quantizer both require a search through lookup tables. However, there exists a special class of scalar quantizers, called uniform quantizers, that can be designed and implemented much more simply and are thus often preferred in practice to optimal scalar quantization.

In a *uniform scalar quantizer*, the subintervals of the quantizer are all of the same size, quantizer stepsize q, where q is given by the range of the input x and the number of quantizer levels:

$$q = \frac{x_{\max} - x_{\min}}{N}. \tag{3}$$

Thus, the range of the input data is partitioned into *uniform* intervals of the same size. Uniform quantizer Q is implemented as

$$Q(x) = \left\lfloor \frac{x - x_{\min}}{q} \right\rfloor,$$

where $\lfloor \cdot \rfloor$ denotes the floor operation. The index set is then

$$I = \{0, 1, 2, \ldots, N-1\}.$$

The corresponding inverse quantizer outputs the midpoint of each interval:

$$\begin{aligned} Q^{-1}(i) &= i \cdot q + \frac{q}{2} + x_{\min} \\ &= i \cdot q + c, \end{aligned} \tag{4}$$

4

where $c = \frac{q}{2} + x_{\min}$ is a constant. Note that while uniform quantizer Q is nonlinear as in the case of a general scalar quantizer, inverse quantizer Q^{-1} is an *affine* operation in the uniform case. Below, we exploit this affine property to perform uniform inverse quantization within the graphics pipeline of a graphics rendering system without additional computation. As a final observation, we note that uniform scalar quantizers are optimal (in the sense of Eq. 1) only when the input x is uniformly distributed between $x_{\min}$ and $x_{\max}$; i.e., when $p(x) = (x_{\max} - x_{\min})^{-1}$. Next, we review previous use of quantization in geometric models as it has appeared in prior literature.

3. Quantization in Geometric Models

The data contained in a geometric model, e.g., the vertex coordinates, vertex normals, etc., are ideally infinite-precision real numbers. However, in order to use this data in a digital computer, these real values must be quantized to a finite precision. As argued above, this truncation of precision is always necessary when dealing with real-world quantities in the discrete domain of computers, since even a single infinite-precision number would theoretically require, in general, an infinite amount of storage space. In order to restrict storage space to reasonable sizes, real numbers are usually represented in computers as single- or double-precision floating point numbers; these floating-point representations are *implicit* scalar quantizations of the corresponding infinite-precision real numbers (note that the "scalar quantizers" implicit in standard floating-point representations are not uniform). Of course, the specific form of the 32- or 64-bit floating-point representations used by modern computers has more to do with the hardware aspects of computer architecture than the nature of the data being represented. Indeed, from the viewpoint of information theory, the choice of 32- or 64-bit scalar quantization is arbitrary. As discussed below, previous authors have made essentially this same observation and have proposed techniques for quantization in geometric compression, suggesting that geometric models can be adequately represented using significantly fewer bits per floating-point number.

3.1 Previously Proposed Geometric-Compression Quantizers

Deering [?], and subsequently Taubin and Rossignac [?], proposed the following *ad hoc* approach to quantizing the x-, y-, and z-coordinate values of vertices in a geometric model. First, the coordinate values are normalized to a unit cube, and then the normalized coordinates are "rounded" to fixed-length integer values. Clearly this operation is equivalent to quantizing each coordinate with a separate scalar quantizer which technically may or may not be uniform, depending on how the rounding operation is performed. Due to the final representation of the coordinates as integers, the number of quantizer levels used in this approach is restricted to be a power of 2. For example, suppose that the coordinates are rounded to k bits; the equivalent scalar quantizer has $N = 2^k$ levels. Deering [?] has suggested that quantization to 16 or fewer bits ($k \leq 16$) is sufficient for the xyz coordinate positions of most geometric objects.

Deering's suggestion of 16 bits as an upper bound on tolerable vertex-position quantization was motivated by visual observations and circuit-implementation considerations and has come to represent somewhat of a "rule of thumb" for the use of quantization in geometric compression. However, only the quasi-uniform, rounding approach to quantization was considered in Deering's work. Indeed, analysis of other more general forms of quantization, notably Lloyd-Max optimal quantization, is absent from prior literature. The main contribution of this paper is a more rigorous approach to scalar quantization that is both general and amenable to hardware-based decoding. In particular, our quantizer is not restricted to having the number of quantizer levels be a power of 2—more general values of N may aid the later incorporation of variable-length coding techniques (e.g., Huffman or arithmetic coding) into the geometric-compression framework, although we do not explicitly consider such approaches here. Finally, the results presented later give evidence that our approach to uniform scalar quantization works just as well as the optimal scalar quantizer without requiring the additional computational overhead associated with optimal-quantizer design and implementation. Below, we outline our proposed quantization technique and describe its implementation in a

5

terrain-rendering system.

3.2   Proposed Approach to Quantization for Geometric Compression

The key to reducing memory use through geometric compression while not incurring overhead due to decompression is to place the decompression operation in the hardware rendering pipeline. Whereas Deering [**?**] proposes the addition of a specialized hardware decompression engine to the traditional graphics rendering pipeline, our approach to quantization capitalizes on the hardware capabilities common to graphics subsystems in modern workstations.

It was observed above that the inverse quantizer for uniform scalar quantization as implemented in Eq. 4 is an affine transformation. As such, it can be implemented in a 4×4 matrix using homogeneous coordinates. Specifically, suppose we have quantizers Q_x, Q_y, and Q_z for the x, y, and z vertex coordinates, respectively, of a vertex. The inverse quantizers from Eq. 4 are

$$\hat{x} = Q_x^{-1}(i_x) = i_x \cdot q_x + c_x$$
$$\hat{y} = Q_y^{-1}(i_y) = i_y \cdot q_x + c_y$$
$$\hat{z} = Q_z^{-1}(i_z) = i_z \cdot q_x + c_z.$$

In turn, these inverse quantization operations can be performed simultaneously by the following homogeneous matrix equation

$$\begin{bmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \\ 1 \end{bmatrix} = \begin{bmatrix} q_x & 0 & 0 & c_x \\ 0 & q_y & 0 & c_y \\ 0 & 0 & q_z & c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_x \\ i_y \\ i_z \\ 1 \end{bmatrix}. \tag{5}$$

Most desktop graphics-accelerated computers include hardware to handle the geometric transforms of translation, scale, and rotation common to interactive 3D applications through the use of homogeneous matrices. Because inverse uniform scalar quantization can be expressed in a homogeneous matrix, this hardware can be exploited to eliminate computational overhead that would ordinarily have been required to decode the data to their original values. A separate decoding operation is not necessary; the object-space-to-world-space homogeneous transformation matrix can simply be multiplied by the 4×4 matrix in Eq. 5 and the resulting matrix can be passed to the rendering pipeline. Because of this fact, not only can the data be stored in its quantized form in memory but the quantized values can be passed directly through the rendering pipeline without alteration. Moreover, the rendering pipeline does not need any specialized hardware as the decoding is performed automatically as part of the object-space-to-world-space transformation. Below, we explore the use of this proposed approach to quantization within our application of interest, terrain rendering.

4.   Terrain-Rendering System Details

4.1   Iris Performer

Iris Performer is a graphics library provided by Silicon Graphics, Inc., designed for high-performance, interactive, multi-processor applications. It also provides two methods for interactive viewing of terrain datasets, one of which is a simple algorithm based on subdivision of the terrain dataset into small manageable tiles. For each tile, several levels of detail are generated, requiring that each tile not only be square but also have dimensions that are a power of 2. In order to prevent seams along the edges of the tiles, the edges remain at full resolution for each LOD, guaranteeing that tile edges match exactly the edges of adjacent tiles. During rendering, the bounding box of each tile is tested against the view volume for that frame. If the bounding box is determined to be within the view volume, an appropriate LOD is chosen based on the

distance from the view point, and the tile is rendered at that LOD. View culling, LOD computation, and data storage are each handled by existing mechanisms within the Performer scene-graph hierarchy.

Because of its simplicity, tiled rendering does not require much precomputation. Therefore, the terrain datasets can remain in their original file format and be converted into the necessary structures at run time. This approach is contrary to that taken by some terrain-rendering methods which require a large amount of precomputation and possibly the creation of large files on the system. The only requirement of the tile-based approach is that a rectilinear grid of height points (i.e., a DEM) be provided; unfortunately, the data is extremely sizeable once it has been loaded into memory and converted into the tile-based structure. This problem is exacerbated by the fact that Performer does not exploit its requirement for the data to lie on a rectilinear grid. Although the x and y coordinates could be easily computed or stored in a look-up table, the algorithm instead stores not only the x, y, and z coordinates for each point in the dataset for each LOD within each tile, but also the connectivity for the points of each triangle. This method of data storage results in a large amount of redundancy and memory waste.

4.2 Tiling and Quantization

In order to reduce the storage space required by Performer, we code terrain datasets using the scalar-quantization approach discussed above on the terrain height field (z coordinates), designing a different scalar quantizer for the height values of each tile of data. Thus, the Q_z quantizer for a particular tile is adapted to the dynamic range of the height data in that tile. In general, the number of quantizer levels may also vary from tile to tile; however, for simplicity we use the same number of levels for each tile.

In addition to quantization of height values, for even greater compaction, we also apply scalar quantization on the x- and y-coordinate values. As mentioned above, the tile-based rendering mode of Performer requires that the x and y coordinates of each vertex lie on a rectilinear grid, although Performer essentially ignores this structure at data input, expecting all three coordinate values to be provided to the graphics pipeline. Clearly a significant reduction in the size of the dataset can be easily obtained by storing only the z value (terrain height) of each vertex and not the x and y coordinates, which lie on the rectilinear grid. The x and y coordinates, because of their rectilinear structure, can be easily reconstructed for each frame during rendering. This approach would require that the tiles be built for each frame; however, as opposed to performing large computations for the x and y coordinates through software, the hardware rendering pipeline can again be employed through the use of a homogeneous matrix.

Specifically, suppose that the data for the current tile is arranged on an $M \times M$ rectilinear grid. Then, x- and y-coordinate quantizers can be defined as

$$q_x = \frac{x_{\max} - x_{\min}}{M - 1}$$

$$q_y = \frac{y_{\max} - y_{\min}}{M - 1}$$

$$c_x = x_{\min}$$

$$c_y = y_{\min}$$

$$Q_x(x) = \left\lfloor \frac{x - x_{\min}}{q_x} \right\rfloor$$

$$Q_y(y) = \left\lfloor \frac{y - y_{\min}}{q_y} \right\rfloor$$

Using the above values of q_x, q_y, c_x, and c_y, inverse quantization for the x and y coordinates can be computed via the 4×4 homogeneous matrix of Eq. 5. Now, instead of storing any values for x and y

coordinates, the algorithm can, on the fly, pass integers 0 to $M - 1$ into the rendering pipeline, automatically producing the proper x and y coordinates for the tile with minimal overhead in building the tiles of each frame. Clearly the reconstruction of the x and y coordinates occurs simultaneously with the inverse quantization of the z height values within the graphics pipeline. The storage space required for each tile then consists merely of a two-dimensional array of quantized height indices, $i_z(x, y)$, and information to construct the inverse-quantization matrix, namely, q_x, q_y, q_z, c_x, c_y, and c_z.

Determining which tiles need to be rendered is a simple process given knowledge that the terrain is merely a two-dimensional array of height values. Using the x and y location of the view point, the tile immediately below the view point can be quickly determined. Tiles adjacent to the current tile can be tested to determine if they should be built by a simple test for intersection between the bounding box of the tile and the view volume.

LOD methods can be used to reduce the number of triangles to be rendered during each frame. A simple approach is to require that tiles have a dimension of $M = 2^\ell + 1$ for some integer ℓ and to choose an LOD determined by the distance of the tile from the view point. Tiles for a specified LOD are constructed by generating a triangular mesh with the dimensions of 2^L where L ranges from 1 (the lowest LOD) to $\log_2(M - 1)$ (the highest LOD).

4.3 Stitching

The proposed approach to quantization outlined above must be modified slightly along the edges between tiles; otherwise, the vertices lying on the edge of a tile would not usually match corresponding vertices on the edges of adjacent tiles, since each of tile is quantized independently of the other tiles. That is, suppose two neighboring tiles contain a vertex lying on the edge of both tiles. Because these tiles may use different quantizers Q_z, it would be likely that the quantized heights $\hat{z}$ of the vertex would be different from one tile to the other. This difference could result in visible seams or gaps in the midst of the terrain. In order to prevent this loss in visual quality, a small strip of a unit width between each of the tiles is rendered apart from the process we have outlined above (see Fig. 1). Unfortunately, we can no longer rely on the rendering-pipeline to invert quantization during this stitching operation. Since the strip between two tiles shares vertices with both of the neighboring tiles, and these tiles may have different inverse-quantization matrices (due to having different height quantizers Q_z), one transformation matrix is no longer sufficient for all the vertices of the triangles of the stitching strip. However, as most hardware systems that accelerate matrix transformations do not allow the loading of a new transformation matrix while a geometric primitive is being defined, inverse-quantization calculations must be done in software for each of the vertices in the strip.

In light of the discussion above, the stitching operation of our system operates as follows. The quantization of the x, y, and z coordinates of each vertex in the stitching strip is inverted in software and the resulting points are then used to generate triangles creating a seamless transition from tile to tile. In addition to compensating for differing quantizers from tile to tile, this triangulation also takes into account differing levels of detail between adjacent tiles. Because the stitching-strip geometry is already organized in the form of a strip, we pass triangle strips to the rendering hardware to reduce the number of redundant vertices that are sent down the rendering pipeline.

5. Results

In this section, we present some results that illustrate the tradeoff involved in using techniques proposed above. Our approach to quantization as applied to a terrain dataset offers significant reduction in the memory required to store the dataset during rendering. In our terrain rendering application, the original dataset representation used $3 \times 64 = 192$ bits per vertex to represent the x, y, and z coordinates of a vertex as floating-point values. Our quantized representation stores merely the quantization index for the z coordinate (terrain height) plus a negligible amount of overhead to specify x, y, and z quantizers for each tile. We
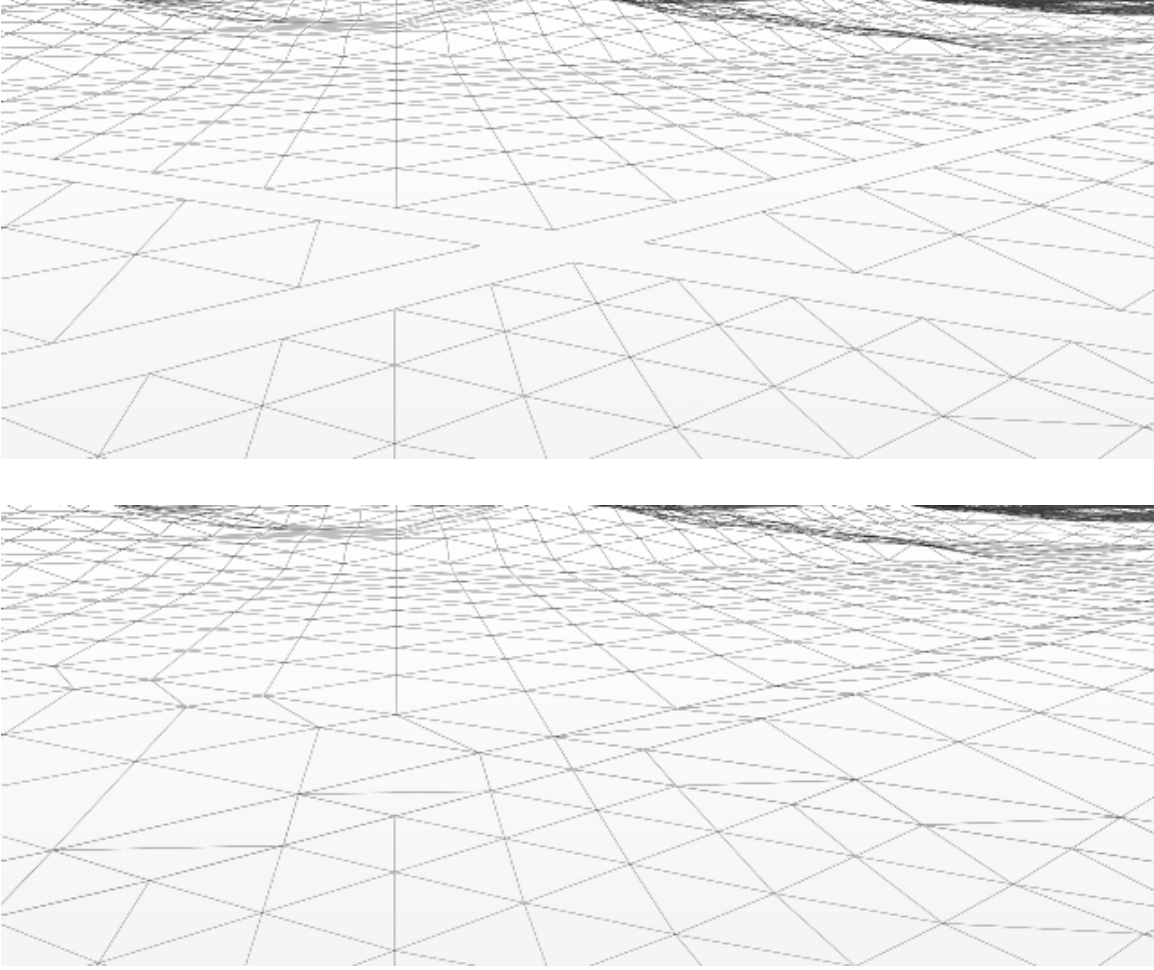
Figure 1: Stitching between adjacent tiles. Stitching compensates for differing height quantization and differing levels of detail between adjacent tiles.

restrict our attention here to the case that a fixed-length binary code is used to represent each quantization index; as a consequence, the number of quantization levels of Q_z is $N = 2^k$ where k is the number of bits per vertex used in the quantized dataset. Therefore, the *rate* of the dataset is approximately $R \approx k$ bits per vertex. Although our approach to quantization allows the Q_z quantizer to vary from tile to tile, to simplify the discussion here, we use the same number of quantizer levels N for all tiles of the dataset.

The rate as defined above describes the compression efficiency of the quantized representation. In order to measure the quality of the representation with respect to the original dataset, we treat the field of terrain heights as a digital image and employ a traditional distortion measure commonly used in image-processing applications. Specifically, we approximate the average distortion of Eq. 2 with the mean square error (MSE) between the quantized and original height fields:

$$D_{\mathrm{MSE}} = \frac{1}{K^2} \sum_{\forall i} \left(z_i - Q(z_i) \right)^2 ,$$

where z_i is the height of a vertex in the dataset, and the dataset contains $K \times K$ vertices. We then express the

9

distortion in terms of peak-signal-to-noise ratio (PSNR) in decibels (dB), as is common in image processing:

$$D_{\mathrm{PSNR}} = 10 \log_{10} \frac{\left(\max_i(z_i) - \min_i(z_i) \right)^2}{D_{\mathrm{MSE}}}.$$

In Fig. 2, we plot the distortion performance vs. quantizer rate of our proposed approach to quantization as obtained for a terrain dataset of the western United States. For comparison purposes, we give results for the case in which the quantization used for the height values (i.e., Q_z) is optimally designed for the dataset, in addition to results for our proposed approach, which considers only uniform scalar quantization. In this latter case, the graphics pipeline automatically reconstructs the height field via the inverse quantization of Eq. 5 while rendering the quantized dataset, and the design of the quantizer involves merely an application of Eq. 3 to the original z height values as a preprocessing step. On the other hand, for the optimal quantizer, the design process is quite computationally complex, requiring up to 30 minutes to find the optimal quantizer stepsizes q using the iterative Lloyd-Max algorithm. In addition, due to the nonuniform nature of the optimal quantizer, the inverse quantization must be performed by a software-based search through a table of quantizer intervals and their corresponding intervals rather than by exploiting the hardware rendering pipeline.

From Fig. 2, we observe that the optimal quantizer offers a modest improvement in PSNR over the uniform quantizer at most of the rates considered. The question at hand then is whether this amount of gain is significant, given that the ultimate gauge of the quality of the reconstructed terrain dataset is not PSNR measurements between the reconstructed and original height fields, but rather the visual quality of the rendered output. In Figs. 3 through 6, we show a scene rendered from the reconstructed terrain for a variety of quantizer rates for both uniform and optimal quantization strategies. This scene has a mixture of both flat-land and ridge terrain features. For quantization under 6 bits per vertex, distortion of the terrain is quite visible—ridge and mountain features exhibit sharp variations in slope while small undulations in the terrain are lost. This distortion occurs for both quantization strategies, with the optimal quantizer perhaps producing slightly better looking results at very low rates (2 and 4 bits per vertex). However, very little difference is evident between the images for quantizer rates of 6 bits and over. In addition, the images for both strategies are essentially visually lossless for these higher rates when compared to the original (non-quantized) terrain. Similar results were observed for other regions in the terrain, suggesting that perceptually near-lossless terrain rendering is possible at around 6 to 8 bits per vertex, corresponding to a compression ratio of 24:1 to 32:1 in comparison to the size of the original dataset representation (192 bits per vertex). Additionally, we observe that, although the optimal quantizer produces slightly better PSNR, for useful rates (for which distortion is not visually significant, 6 to 8 bits per vertex and above), no better visual performance is obtained, despite the extra computation involved in designing the optimal quantizer and decoding the optimally quantized representation.

Finally, we observe that Deering's "rule of thumb" of 16 bits per vertex [**?**] as an upper bound on tolerable quantization rate is perhaps rather loose. Our observations indicate that much greater compression, on the order of 6 to 8 bits per vertex, is visually lossless in situations we have considered. This is particularly true for common rendering applications involving terrains, such as fly-throughs and browsing. In our application, fly-throughs using 6-bit quantization yielded animations of high quality; sample animations in MPEG format are available at `http://www.erc.msstate.edu/labs/vail/projects/terrain_render.html`

6. Conclusions

In this paper, we presented encoding techniques for representing vertex coordinates of a terrain model. In particular, we examined the suitability of uniform and optimal quantization. Uniform quantization has the advantage of being amenable to incorporation in the hardware graphics-rendering pipeline as inverse
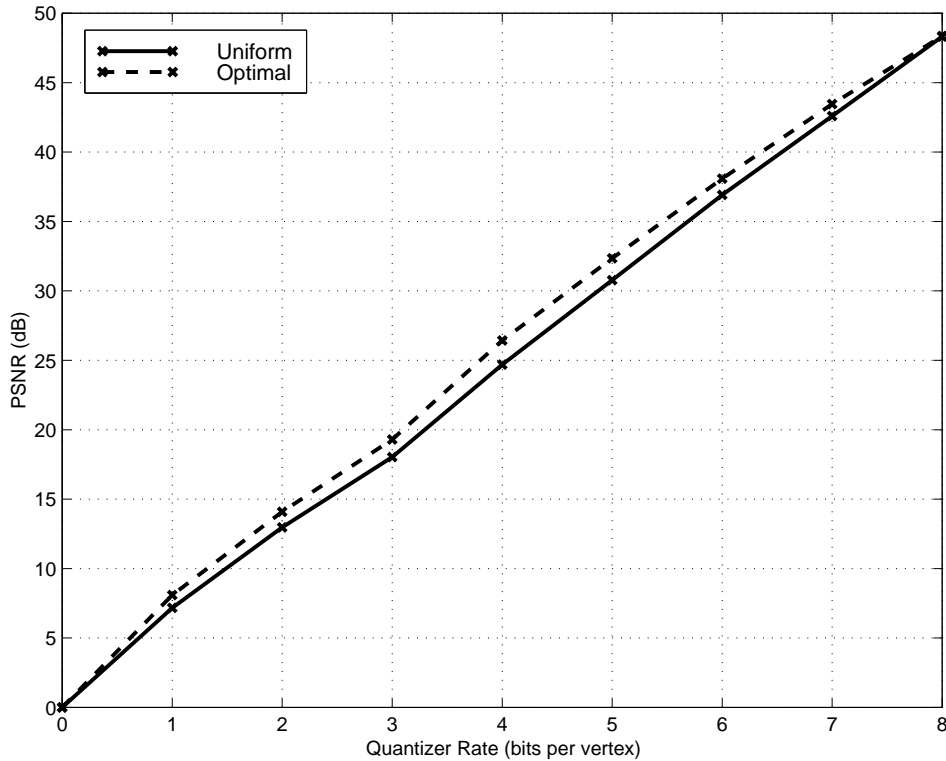
Figure 2: Rate-distortion performance of the proposed approach to quantization.

quantization can be achieved there via a simple matrix operation. Thus, our simple technique bridges the gap between efficient data representation and timely rendering of very large terrain data. In our case, the rendering of a very large terrain which could not otherwise be ingested into the memory subsystems of many commercial graphics hardware was made possible. In addition, our results argue that uniform quantization provides the same level of accuracy as the optimal-quantization scheme for terrain datasets. Surprisingly, we found that a 6-bit quantizer suffices for datasets we considered, much fewer bits than the 16 bits of quantization that Deering [?] observed as being sufficient for viable geometric encoding. Although we focused on terrain models here, our technique can form the trailing end of any polygonal simplification technique, as quantization plays a fundamental role in the larger paradigm geometric compression. Finally, the work presented here can be viewed as providing a rigorous link between the precision requirements of modeling and those of rendering.

In the future, we intend to explore the addition of other compression and coding techniques to the approach outlined here to allow very succinct storage on hardware disks or expedient transmission through low-bandwidth networks. We are particularly interested in the design of encoding techniques that both preserve geometrical properties of the terrain and exploit the underlying hardware rendering pipeline. We intend to seek answers by resorting to appropriate rate-distortion techniques that are in vogue in the image-processing community.
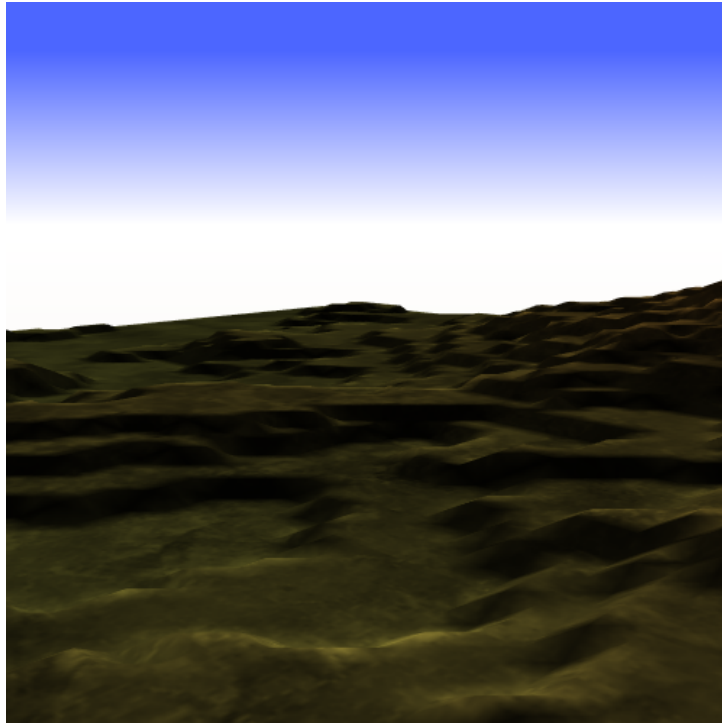
11
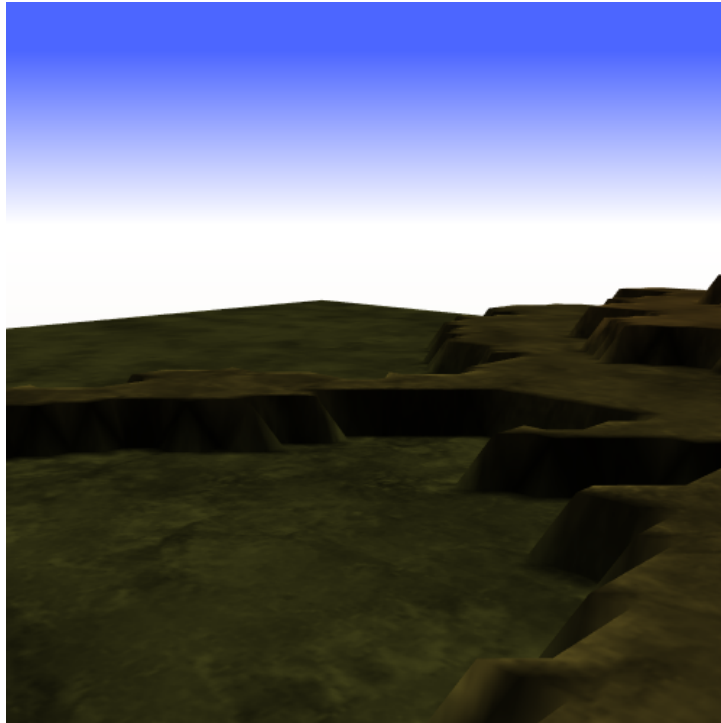
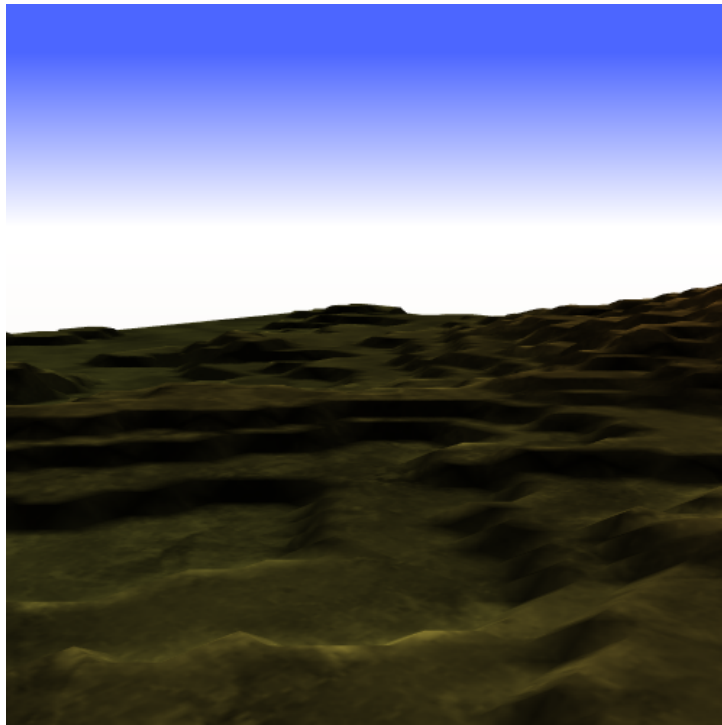(a) Uniform scalar quantizer
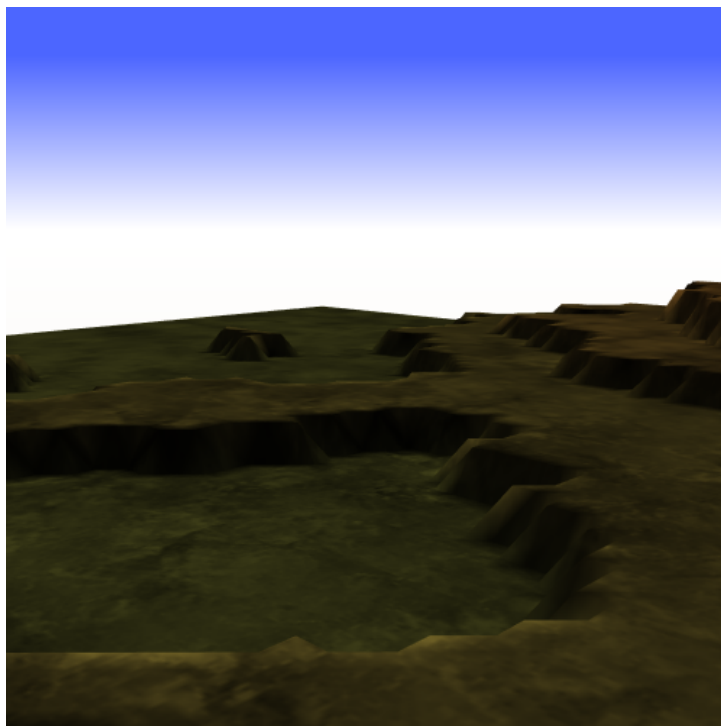
(b) Optimal scalar quantizer

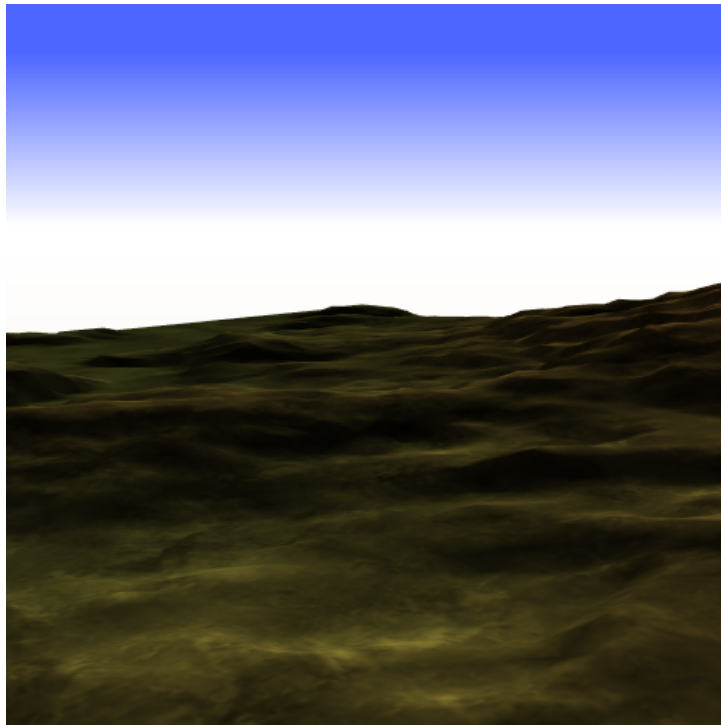**Figure 3:** Reconstructed scene, quantizer rate = 2 bits per vertex
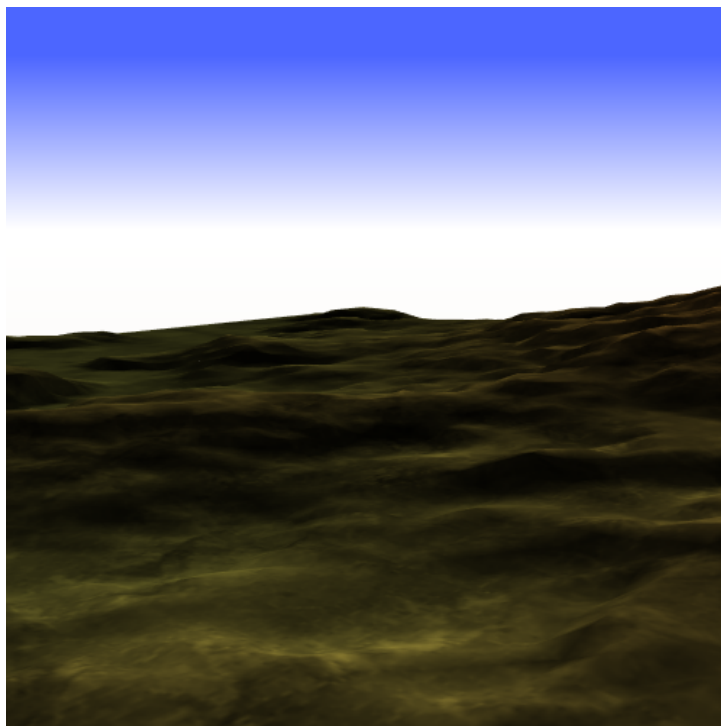
(a) Uniform scalar quantizer

(b) Optimal scalar quantizer

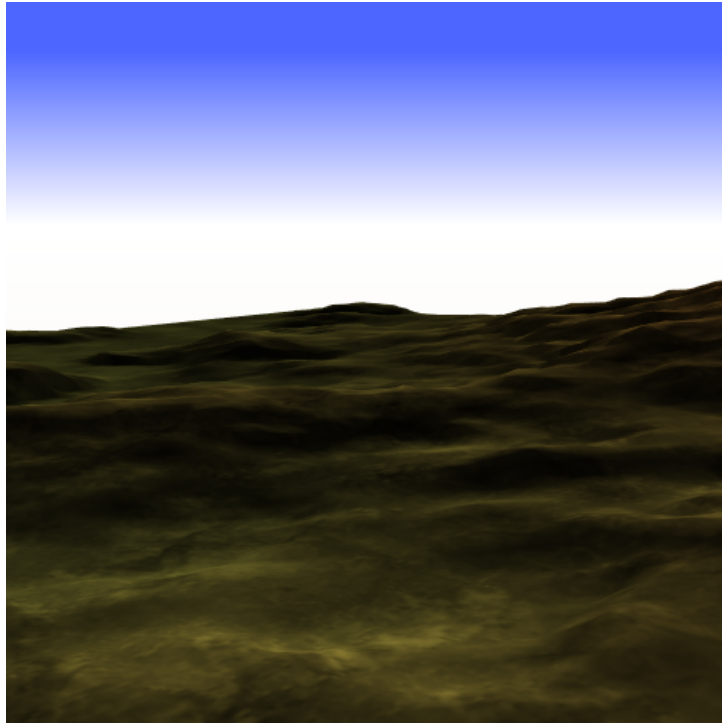**Figure 4:** Reconstructed scene, quantizer rate = 4 bits per vertex
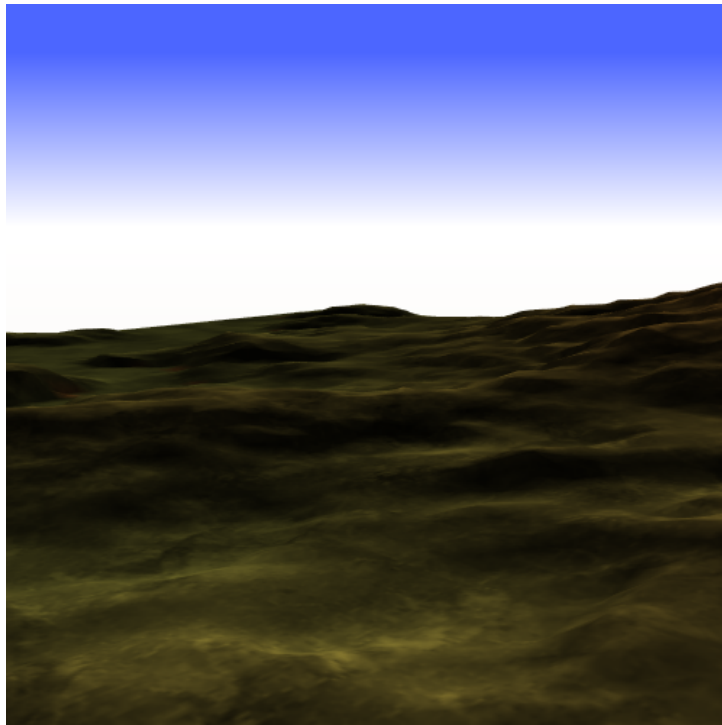
(a) Uniform scalar quantizer

(b) Optimal scalar quantizer

Figure 5: Reconstructed scene, quantizer rate = 6 bits per vertex

(a) Uniform scalar quantizer

(b) Optimal scalar quantizer

Figure 6: Reconstructed scene, quantizer rate = 8 bits per vertex