Compressed Facade Displacement Maps

Saif Ali, Jieping Ye, *Member*, *IEEE*, Anshuman Razdan, *Member*, *IEEE*, and Peter Wonka, *Member*, *IEEE*

Abstract—We describe an approach to render massive urban models. To prevent a memory transfer bottleneck, we propose to render the models from a compressed representation directly. Our solution is based on rendering crude building outlines as polygons and generating details by ray-tracing displacement maps in the fragment shader. We demonstrate how to compress a displacement map so that a decompression algorithm can selectively and quickly access individual entries in a fragment shader. Our prototype implementation shows how a massive urban model can be compressed by a factor of 85 and outperform a basic geometry-based renderer by a factor of 40 to 80 in rendering speed.

Index Terms—Massive urban models, displacement mapping, real-time rendering, compression.

1 Introduction

In this paper, we describe an approach to render massive urban models requiring several gigabytes of data. It is especially difficult to render fly overs, where almost the complete model is visible, and the visual portion of the scene requires more memory than is available on the graphics card and in the main memory. As a solution, we present a framework that renders from a compressed representation directly and avoids slow memory transfers from the disk to the main memory and GPU at every frame.

We assume that building facades are stored as large polygons and facade details are stored as displacement maps and material index maps. A displacement map is a matrix $D \in R^{m \times n}$, where each entry represents a displacement from a reference plane, and a material index map is a matrix $M \in N^{m \times n}$, where each entry references a texture in a texture library. To render this representation, the large facade polygons are rasterized to initialize a fragment shader that computes an exact ray-surface intersection (see Fig. 1) using ray casting. The idea of the fragment shader ray caster is to march a ray over the displacement map until a surface intersection is found. While a naive algorithm uses a large number of very small steps, it is often beneficial to use acceleration data structures to allow the ray to jump larger distances in regions of free space. Two prominent examples of acceleration data structures are cones [1] or spheres [2] of free space. These acceleration data structures are stored as an additional matrix A (or several additional matrices). We call the matrices D, M, and A the detail maps of a facade.

In this paper, we will present a rendering framework that precomputes compressed detail maps and decompresses individual matrix entries in the fragment shader. We need to tackle two challenges. First, we have to find a suitable compression algorithm. Second, we have to adapt current fragment shader ray-tracing algorithms. The solutions to these two problems are the major contributions of this paper:

- We present a novel lossless compression algorithm for facade detail maps. While the combination of compression and fragment shader ray tracing is a novel approach by itself, we additionally improve the best previously used algorithm, the singular value decomposition (SVD). The SVD was used in different contexts, for example, to compress BRDFs [3]. Our results show that our representation is 2.5 to 8.6 times smaller and takes a shorter time to decompress (resulting in faster rendering speed). The main idea of our approach is to factor a matrix using binary values instead of floating-point values.
- We analyze existing fragment ray-tracing techniques and propose a new acceleration data structure A: we store indices of boxes of free space for each entry in D. We need to adapt previous work 1) because of visual quality and efficiency issues connected with rendering discontinuities and 2) because previous acceleration data structures do not compress well and therefore do not fit into our framework. While our new data structure and the rendering algorithm are not very sophisticated, the explanation of algorithm details and the connection between acceleration data structure design and compression ratios is important for understanding the overall framework. We will show that our rendering times are a factor of three to seven faster than a naive combination of previous work in displacement mapping (relief mapping) and compression (SVD).

We call our new representation CFDM for *Compressed Facade Displacement Map*. For massive model rendering, our

- *S. Ali is with AMD, 2145 3rd street, Santa Clara, CA 95054. E-mail: mail.saifali@gmail.com.*
- *J. Ye is with the Department of Computer Science and Engineering, Arizona State University, Box 878809, Tempe, AZ 85287-8809. E-mail: jieping.ye@asu.edu.*
- *A. Razdan is with the Division of Computing Studies, Arizona State University, Mail Code 0180, Tempe, AZ 85287-5906. E-mail: razdan@asu.edu.*
- *P. Wonka is with Arizona State University, 342 Byeng, Brickyard Building, 3rd Floor, 699 S. Mill Avenue, Tempe, AZ 85281. E-mail: pwonka@gmail.com.*

Manuscript received 8 Sept. 2007; revised 6 May 2008; accepted 2 July 2008; published online 17 July 2008.
Recommended for acceptance by P. Dutre.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-2007-09-0135.
Digital Object Identifier no. 10.1109/TVCG.2008.98.

Fig. 1. A box rendered with compressed facade displacement maps.

results will show that we can render a city of 30,000 buildings in real-time rates using a memory footprint of 1.62 Gbytes at 30-60 fps, depending on the screen size. Geometry-based rendering would require 23-Gbyte memory for 180 million triangles and corresponding textures rendered in about 0.5 fps. While our approach gives very strong results for typical building facades, the compression requires detail maps (matrices) that can be broken down into larger submatrices that have identical entries. The extension of CFDM to general displacement maps D is future work.

2 Related Work

We structure and review related literature in the following groups: polygonal height field rendering, ray tracing of height fields, displacement mapping, fragment shader ray tracing, and matrix factorization. While we use massive model rendering as the motivational framework, our contribution is not related (and mostly orthogonal) to system papers describing level-of-detail, occlusion culling, and memory management strategies for massive models. We therefore chose not the review these techniques in detail but refer to two recent papers that include a nice overview of the literature [4], [5].

Polygonal rendering. Many height fields, e.g., terrains stemming from geospatial data sets, are sampled on a regular grid. This data structure is identical to a classical displacement map. The focus of research in terrain rendering is to compute triangular levels of detail to limit the number of polygons selected for rendering [6], [7], [8].

Traditional ray tracing. There are many algorithms to compute a ray intersection with related data structures such as height fields [9], [1], [10], [11] or displacement mapped triangle meshes [12]. The approach closest to our work is described in [1] and [10].

Displacement mapping. Cook introduced displacement mapping [13] as an improvement over bump mapping [14] to render detailed surfaces with correct silhouettes. Several approaches were proposed for rendering displacement mapped geometry by on-the-fly retessellation of a base mesh using specialized hardware architectures [15], [16], [17]. These necessitated alterations in the standard graphics pipelines or the installment of dedicated displacement mapping units.

Fragment shader ray tracing. There are two important parts to traditional displacement mapping. The first part is to set up bounding prisms placed over the base triangular mesh [18] and how to consider surface curvature. This part is straightforward for our application as we currently only consider flat facades. The second part is to find efficient ray displacement map intersection strategies. Previous papers propose several interesting root finding methods, e.g., [19] and [20], and acceleration data structures [2], [21], [22], [23], [24] to help the intersection computation. In this paper, we propose a method that outperforms previous approaches for our specific application but sacrifices generality so that it is not suitable to render general displacement maps considered by other authors. Our general compression framework, however, could be used in conjunction with other methods.

Matrix factorization in computer graphics. Most successful compression strategies for real-time rendering are related to matrix factorization. Matrix factorization is a powerful tool with many applications in computer graphics. The most popular method is the SVD [25], which is commonly used for principal component analysis (PCA) [26]. SVD allows us to compute the optimal low-rank approximation of a matrix in terms of both the *Frobenius* norm and the 2-norm. PCA and SVD were used to compress many computer graphics data sets, for example, bidirectional reflectance functions (BRDFs) [3], [27], [28], a view-dependent distance function [23], animation data [29], image databases [30], [31], and precomputed radiance transfer [32]. Early work by Kautz and McCool [3] proposes a simple rank-one matrix approximation scheme to avoid the costly and memory-intensive computation of the SVD for large data sets. Homomorphic factorization allows us to decompose a matrix as the product of multiple matrices. This approach has been explored for the compression of BRDF data [33], [34]. A more general formulation that mixes sums and products is the chained matrix factorization proposed by Suykens et al. [35]. Modifications in the factorization, such as nonnegative factorization (NMF) [36], allows the incorporation of constraints and weights. This is helpful for importance sampling [37], editing [38], and diagonal dominant BSSRDF data [39]. Clustered PCA (called local PCA in machine learning) first partitions data into different clusters and performs PCA locally. Examples of clustered PCA in computer graphics include radiance transfer [40], [41], mesh animations [42], and motion capture databases [43]. In this paper, we will also use SVD for compression. Additionally, we propose a new factorization algorithm that outperforms SVD for our application.

3 Overview of the Method

We first provide the system overview and then describe the two main building blocks of this paper.

System overview. Before a massive model can be rendered, it has to be created. Recent work in industry [44] and academia [45] shows how building textures can be analyzed and segmented so that displacement maps $D \in R^{m \times n}$ and material index maps $M \in N^{m \times n}$ can be created. As it will take a few more years until these models are created in a large scale, we use Photoshop to model a few example facades and replicate them to create a massive model. We use a material library of several wall, glass, and wood materials stored in a texture array. We assume that the facades are characterized by regular, repetitive, and boxlike patterns. The question that we address in this paper is the following: how can we render these building models as efficiently as possible from a compressed representation so that we have high compression rates with only a modest impact on rendering performance?

Facade ray tracing. We investigated several existing fragment shader ray-tracing algorithms and propose a new modification. Instead of storing cones or spheres of free space for each texel in a displacement map, we store indices to boxes of free space. In Section 4, we will explain the details of this acceleration data structure A and the fragment shader ray tracer. We also discuss several alternative design choices that we experimented with and the reasons they were not as successful.

Map compression. The amount of memory required to store the detail maps D, M, and A even for a single facade is significant. Therefore, we propose a solution that interprets all maps as matrices and factors them into the product of three matrices (X, S, and Y). The reconstruction of the original matrices can be computed in the fragment shader for each matrix entry individually. In Section 5, we provide details about the factorization algorithm, the representation, the reconstruction in the fragment shader, and integration with the ray tracer.

4 Facade Ray Tracing

This section describes a new acceleration data structure and the corresponding ray tracing algorithm for facade displacement maps. Our compression and decompression framework can be easily integrated into this fragment shader ray-tracing algorithm, by substituting lookup functions in 2D textures by compressed lookup functions. The key to designing a successful algorithm is to find an acceleration data structure that leads to very few search steps and that is simple enough so that it compresses well. The short explanation of our solution is that we extend the idea of the cone data structure proposed by Paglieroni [1], [10] from cones of free space to indices of boxes of free space. We first analyze previous work in the context of facade displacement maps and then give the details of our algorithm.

The displacement map D is a sampled representation of a discontinuous function $f(u, v) \rightarrow height$ over a rectangular domain $[0, 1] \times [0, 1]$ that is stored in a matrix. In contrast to displacement maps for continuous functions, we are not interested in a smooth reconstruction, as we need to retain the sharp edges prominent in building facades. Therefore, the reconstruction of choice is the box filter. See Fig. 2a for a 2D example displacement map.

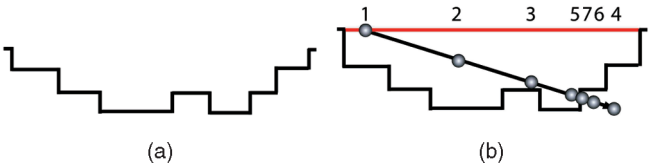


Fig. 2. (a) A sampled facade displacement map is reconstructed with a box filter. Please note the resulting discontinuities. (b) An intersection algorithm can use a combination of linear and binary search to find the intersection of a ray and a facade displacement map. The first four samples in the sequence are linear search steps, and the last three samples result from a binary search.

4.1 Analysis of Previous Work

A very simple method to compute the intersection of a displacement map and a ray is to use a large number of linear search steps. On a 1,000 × 1,000 map, this would require at least 1,000 steps. While huge accelerations are possible by using fewer search steps, this strategy simply ignores most of the data. A good optimization in practice is to use a linear search with larger search steps to get an estimate of the location of the intersection point and then use a binary search to compute a more exact intersection. This algorithm was proposed by Policarpo et al. [19] and is shown in Fig. 2. The strength of this algorithm is its simplicity and the fact that the algorithm does not need to store an acceleration data structure. We will use this algorithm for comparison. The main challenge of this algorithm is that it uses unnecessary search steps if used with facade displacement maps. This is especially a problem in compressed rendering where search steps are getting more expensive.

Therefore, we experimented with acceleration data structures to accelerate rendering by space leaping. The general idea of space leaping is to store a bounding primitive of free space for each texel in the displacement map. In recent papers, spheres and cones were proposed as acceleration data structures (see Fig. 4). We experimented with both data structures but dismissed spheres because they require a 3D rather than 2D data structure. Cones need less memory, but we encountered three problems: 1) Near a discontinuity (and we have many), the cones will get very narrow, and the intersection routine becomes inefficient. We found that the intersection computation typically requires 3 to 10 times more steps than our proposed method (please note that this is only true for facade displacement maps and not the general case). 2) Discontinuities cannot be handled well by cone step mapping (and also other previous methods), because the discontinuities are not appropriately textured. A simple solution is to texture discontinuities with the texture of the higher surface. This gives correct results in almost all cases for architectural models, but this modification is difficult to integrate in previous work. Fig. 3 illustrates the importance of discontinuity handling. A related problem is that the all the discussed methods need a separate normal map N to store normals at discontinuities that would require additional memory. 3) The acceleration data structure has little coherence between neighboring texels. This leads to low compression rates and slow decompression times. The overall rendering times of compressed cone step mapping is therefore often slower than compressed relief mapping, because the slow decompression of the acceleration data structure negates the performance gain of space leaping. In

(a) (b)

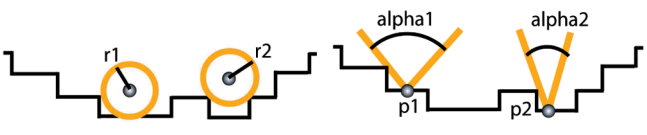
Fig. 3. This figure shows the necessity of handling discontinuities in the displacement map. (a) The combined linear and binary search technique creates noticeable texturing artifacts. (b) Space leaping with boxes can nicely render and texture discontinuities.

the next section, we explain how to modify the ideas from previous work to adapt them to compressed facade displacement mapping.

4.2 Facade Displacement Mapping

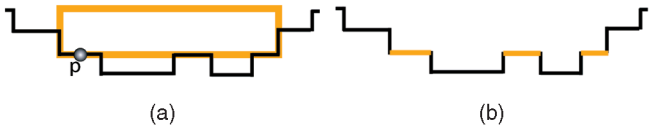
Due to the nature of facade displacement maps, we chose to store a box of free space for each texel in the displacement map. This will allow the ray to jump until it hits a boundary of the box. Fig. 5 shows a free box over a point p. To obtain the best acceleration, we are interested in computing the largest possible box. While the largest box is not always uniquely defined, we do not expect these special cases to have any major impact. Therefore, we use a robust and fast algorithm to make our approach as practical as possible. We use a greedy search that computes the box for a texel t with the following steps: 1) initialize the box to have the size of t, 2) grow the box in all four directions by either one texel if there is free space or zero texels if there is no space, and 3) repeat step 2 until the box cannot grow in any of the four directions. The key part is that all texels with the same height that touch the bottom of that box can share the same box and do not need to start their own computation (see Fig. 5b for all texels that can share a box). Please note that the boxes can overlap (see Fig. 6). An alternative algorithm was proposed in the context of occlusion culling [46]. This algorithm is faster but might produce lower quality results.

Given the maximally free box, we have several options to store the acceleration data structure. The first choice to make is the number of values to be used to describe a box. If we only store one or two values rather than four values, we can only store a conservative approximation, but the algorithm will be simpler and more efficient. For example,



(a) (b)

Fig. 4. This figure shows the idea of using spheres and cones as bounding primitives of free space. (a) The radius of the maximally free sphere is stored for each point in a 3D data structure. Please note that the other bounding primitives only lead to 2D data structures. (b) The opening angle of the maximally free cone is stored for each texel in the displacement map. Please note that multiple geometric quantities can be used to describe the cone.



(a) (b)

Fig. 5. (a) For a point p, the biggest free box over p is shown in orange. The box is represented by the locations of the four side planes to the top, bottom, left, and right in the local displacement map coordinate system. (b) The data structure results in significant coherence. All locations (texels) marked in red share the same values of the acceleration data structure as the point p to the left.

the analog to cone step mapping would be to store the largest square for each texel. While this approach is still faster than cone step mapping, it results in too many search steps and the loss of coherence between neighboring texels. The second question is if we should store the absolute values of the four sides of the box in the facade coordinate system or the relative values as seen from the center of each texel. It is important to note that only the first version works in our framework, because most texels are associated with the same box as their neighbors. This type of coherence is crucial for compression. The third choice was to store one index into a list of four values instead of four values for each texel. By storing indices, we only need to store one entry in A for each entry in D (rather than four entries). Next, we explain the ray-tracing algorithm.

Over each facade, we need to render one quad for the top and zero to four quads on the side to initialize the ray tracing in the fragment shader. A quad is needed for all sides that have a displacement greater than zero. The ray tracer has the following functionality: 1) starting from a rasterized pixel on one of the quads, it has to intersect a ray from the eye through the rasterized position, and 2) at the hitpoint, it has to compute a normal on the fly, look up a material index in the material map, and use the material index to access a material texture in a texture atlas (or texture array on newer graphics cards). We focus our discussion on the ray displacement map intersection. The intersection of a ray with the facade displacement map is computed in tangent space, employing a parametric intersection routine in the fragment shader program. A ray is parameterized by the parameter t, and any point $p(t)$ on the ray is given by

$$p(t) = p_0 + t * \vec{d}, \tag{1}$$

where p_0 is the ray origin, and d is the direction vector.

The viewing ray and direction vector are computed for the four corners of the quad and passed to the fragment stage as a varying parameter. At each ray-tracing step, we look up the acceleration data structure and the displacement map to get to get the four nearest sides and the bottom of the free box over the current texel. The top face is always at zero. These six values define six planes of the box to be

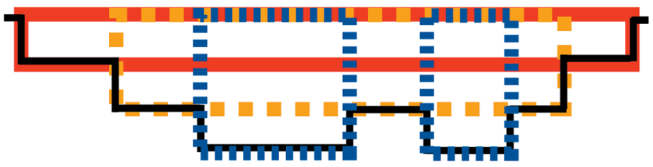


Fig. 6. This figure shows the boxes used in the acceleration data structure. Please note that the boxes can overlap.

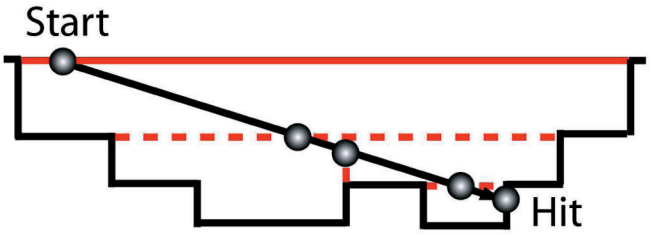


Fig. 7. This figure shows the steps of a ray in the intersection computation. The step size is determined by the acceleration data structure.

intersected. The goal now is to compute which plane the ray hits first. We compute the ray-bounding-plane intersection for every bounding plane producing a set of six values t_i, $1 \leq i \leq 6$. The desired intersection is obtained by choosing the minimum positive parameter value, which also determines the step size for the next step. Fig. 7 shows an example intersection computation in 2D.

Similar to all other ray tracing algorithms, aliasing can become noticeable. We can use texture filtering in the material library and make use of simplified displacement maps to alleviate some of the problem. Further, full-screen antialiasing techniques can also help. Please note that we do not use these techniques in the result section, because it makes comparisons difficult.

5 Facade Compression

The detail maps D, M, and A require several megabytes of data per facade. It is necessary to store higher resolution maps for close-up views. Lower resolution maps would distort the geometry too much and make it impossible to capture details such as window frames.

Please note that we use integer indices for the material index map M and the acceleration data structure A. Therefore, the compression has to be lossless, because similar integer values can index very different entities. This rules out popular lossy compression techniques that use a cosine or wavelet transform (e.g., JPEG). The second important requirement on the compression scheme is that it has to allow for random access to each entry in a matrix. Almost all existing lossless compression algorithms such as zip and rar need to consider larger blocks at once, and it would be impossible to decompress individual elements using random access. Our constraints require that we trade off compression efficiency and decompression speed.

Our approach is inspired by the observation that facades can be modeled based on combinations of function pairs with one vertical function and one horizontal function [47]. See Fig. 8 for a simple example of a facade displacement map D. In this example, all windows and doors are simple rectangles that are displaced by the same value s. The walls are not displaced, and therefore, the corresponding entries in D are zero. The functions x and y are also represented in discretized form as vectors with entries of either one or zero. We can now reconstruct the facade displacement map D using the outer product of the two vectors x and y and scaling the result by s:

$$D = xsy^T. \tag{2}$$

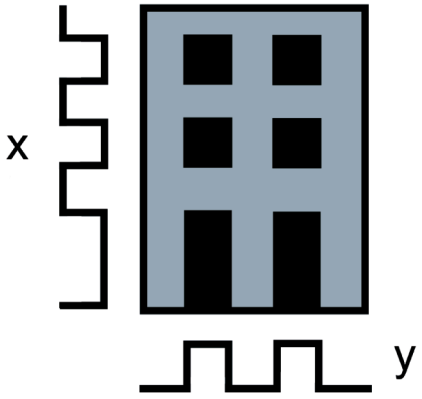


Fig. 8. This figure shows the idea of representing a facade as a combination of vertical and horizontal functions.

The main advantage of this representation is that it allows random access to entries of the matrix. Each entry $D[i, j]$ is computed as the product of three scalar values $x[i] * s * y[j]$. The reconstructed matrix has rank one, as all column vectors are linearly dependent. A more complex displacement map, i.e., a displacement map that corresponds to a matrix of higher rank, cannot be compressed with a scaled outer product of two vectors. For such a displacement map, it will be necessary to use the sum of multiple such outer products. Note that $diag(S)$ stands for a diagonal matrix constructed from vector S:

$$D = x_1 s_1 y_1^T + x_2 s_2 y_2^T + x_3 s_3 y_3^T + \ldots = X * diag(S) * Y^T. \tag{3}$$

To achieve reasonable lossless compression of a matrix as the sum of scaled outer vector products, it is a necessary condition that the matrix has a low rank. This condition is fulfilled by most facade displacement maps due to the dominant rectangular structures.

The efficiency of this type of compression is mainly determined by two different factors. The first factor is the number of outer products we need to reconstruct a matrix. The second factor is the storage requirement per matrix entry.

We propose two methods for factorization explained in the following. First, we will explain how to compute a solution when all elements are restricted to floating-point values. This can be done by computing the SVD of a matrix. Second, we explain a novel factorization algorithm based on binary factorization that gives even better results. The key idea is to restrict the entries of x_i, and y_i to binary values and s_i to floating-point or integer values. This will require more outer products, but each outer product will have significantly less memory requirement. In the following, we will only refer to the factorization of matrices D, but the compression and decompression algorithm for M and A is identical.

5.1 Floating-Point Factorization

If D is an $m \times n$ matrix, then D can be expressed as

$$D = XSY^T, \tag{4}$$

where X is an $m \times m$ matrix, S is an $m \times n$ diagonal matrix, Y is an $nx \times n$ matrix, and X and Y have orthogonal columns. The entries in S are positive or zero and are called the *singular values* of the matrix D. Any matrix D can be

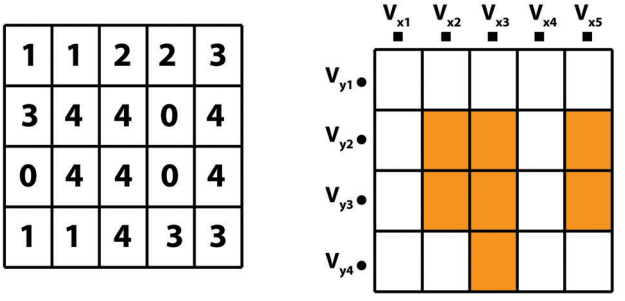


Fig. 9. Left: Input matrix. Right: The binary matrix of the maximum values.

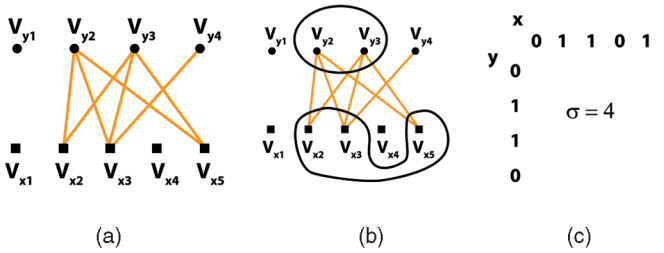


Fig. 10. (a) Bipartite graph corresponding to entries in x and y. (b) Subset of nodes that form a complete subgraph. (c) The rank-one approximation.

written as a weighted sum of outer products of columns of X and Y, where the weights are the singular values s_j [48]:

$$D_{ij} = \sum_{k=0}^{N-1} s_k X_{ik} Y_{jk}. \tag{5}$$

The singular values appear along the diagonal in the matrix S in descending order. We can safely discard the singular values that are zero because the terms in the above outer product corresponding to these singular values will vanish. In our application, we can expect the singular values to go quickly toward zero. To reconstruct the element $D(i,j)_p$, we use the following formula (here, the subscript p indicates the number of singular values retained):

$$D(i,j)_p = \sum_{k=0}^{p-1} X_{ik} S_{kk} Y_{jk}. \tag{6}$$

To guarantee lossless compression for integer entries in M and A, we use as many singular values as are required to guarantee that the rounded reconstructed values are identical to the original values. For D, we keep all singular values that are nonzero. Due to the regularity of facades, there is a clear jump from nonzero to zero singular values.

5.2 Binary Factorization

The idea for the binary factorization is to factor an arbitrary matrix D into the product of three matrices $D = XSY^T$, similar to SVD. However, we want to restrict the entries in X and Y to binary values while allowing floating-point values for the entries in the diagonal matrix S. All previously described factorization methods produce singular values and singular vectors with continuous entries. There are only few exceptions of discrete factorization, e.g., [49] and [50], but previous papers impose restrictions on the matrices that are not useful for the given application. We therefore propose a new numerical algorithm based on two building blocks: First, we want to find the approximation of a matrix as a scaled outer product of two binary vectors x and y. Second, we describe how to use this building block for a complete algorithm.

Rank-one approximation. We first consider the rank-one binary matrix factorization, where for a given matrix $D \in R^{m \times n}$, we find the optimal $x \in \{0,1\}^m$, $y \in \{0,1\}^n$, and σ such that $\sigma x y^T$ approximates the matrix D. In our applications, the entries in D consist of a small set of positive integers. Let $d^D \equiv \max_{ij}\{D_{ij}\}$ be the maximum integer in D. We simply set $\sigma = d^D$. The concept can best be visualized by a binary matrix with entries $\in \{0,1\}$, where all entries that are equal to the highest entry are one, and all others are zero. See Fig. 9 for an example. We then build a bipartite graph $G = (E, V_x \cup V_y)$, where each entry in x corresponds to a node in graph V_x, each entry in y corresponds to a node in the other graph V_y, and E is the set of edges between V_x and V_y. There is an edge between the ith entry in x and the jth entry in y if and only if $D_{ij} = d^D$. For the example in Fig. 9, we show the resulting graph in Fig. 10a. Next, we look for a subset of nodes in V_x and a subset of nodes in V_y so that the subgraph induced by these two subsets forms a maximum complete bipartite subgraph. This is an NP-hard problem [51]. We apply a greedy heuristic for the computation as follows:

Among all the vertices of V_x, we find a vertex u such that the total number of neighbors in V_y ($v \in V_y$ is a neighbor of u if there is an edge between these two nodes) is maximum. We add u to an initially empty set L_x and add all the neighbors of u to another empty set L_y. It is clear that these two sets L_x and L_y form a clique, as all vertices in L_x are connected to all vertices in L_y. Denote $|L_x \times L_y| = |L_x||L_y|$ as the size of the clique formed by L_x and L_y, where $|L_x|$ and $|L_y|$ denote the number of vertices in L_x and L_y, respectively.

We then repeat the above step on the subgraph $\tilde{G}$ of G induced by the neighbors L_y of u, where the vertices of $\tilde{G}$ consist of the ones from both V_x and the current L_y. That is, among the vertices V_x not in the current L_x, we find a vertex $\tilde{u}$ such that the total number of neighbors in L_y is maximum and add $\tilde{u}$ to L_x. The set L_y will also be updated by removing all vertices that are not neighbors of $\tilde{u}$. And so on, until at some point the current set L_y becomes empty. At this point, we get a sequence of cliques formed by different sets L_x and L_y. We find the one with the maximum size $|L_x \times L_y|$ and output the vector x from L_x and the vector y from L_y. See Figs. 10b and 10c for an example.

Rank-p **approximation.** The idea of the complete algorithm is the following iterative algorithm: 1) compute a rank-one binary factorization, 2) subtract the rank-one solution from the matrix, and 3) if the algorithm is not converged go to step 1. The formula for a rank-p binary matrix factorization is given by

$$D_p = \sum_{i=1}^{p} \sigma_i x_i y_i^T, \tag{7}$$

where (x_k, y_k, σ_k) can be obtained by computing the rank-one binary factorization of the residue matrix:

$$D - \sum_{i=1}^{k-1} \sigma_i x_i y_i^T = D - D_{k-1}. \tag{8}$$

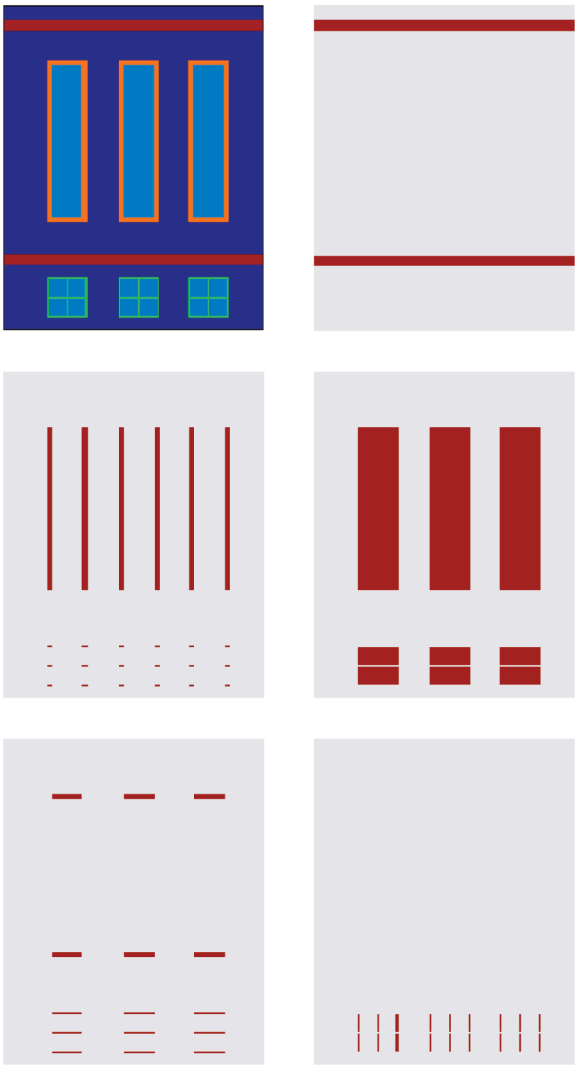


Fig. 11. This figure shows the approximation of a facade displacement map (top left) as a scaled sum of five outer products of binary vectors. The illustration focuses on the structure and does not show the scale of the outer products.

Fig. 11 shows the factorization of a facade displacement map. The algorithm is guaranteed to converge, because each iteration of the algorithm provides an exact reconstruction of at least one more matrix entry. The worst case input to our algorithm is an $m \times n$ matrix where each entry is unique. Then, we will need $m * n$ outer products to reconstruct the matrix.

6 GPU Reconstruction

The reconstruction of compressed matrices using SVD and binary factorization is similar. Here, we describe the details for the binary representation, because it is slightly more involved. The binary matrix factorization algorithm produces three matrices (just like the SVD) with the important difference that elements X and Y are restricted to being either zero or one. While we would use 32-bit floating-point luminance textures of sizes $m \times p\,(X)$, $n \times p\,(Y)$, and $1 \times p\,(S)$ for SVD-based factorization, we need to pack values when using binary factorization. We can consider blocks of 32 consecutive elements in each row of a matrix and pack them inside a 32-bit integer. Thus, each texel finally contains 32 elements from the original matrix, and the matrix is reduced by a factor of 32 in one dimension. For this, we assume that the number of columns in matrices X and Y is an exact multiple of 32. If not, it can be padded with zeros. The OpenGL texture format used is GL_LUMINANCE32UI_EXT. The reconstruction of a single value $D(i, j)$ is obtained by componentwise multiplication of three vectors. The first vector is the row i of X, the second vector is the diagonal of S, and the third vector is the row j of Y. We give the implementation details and code in the Appendix for easier reconstruction of our results.

TABLE 1
Memory Consumption for Four Facades

	NOCOMP	WINRAR	SVD	BIN
Facade1				
Heightfield	586	4	63	8
Material	586	4	63	5
Acceleration DS	2344	7	150	96
Total	3516	15	276	109
Facade2				
Heightfield	3072	7	195	30
Material	3072	7	195	30
Acceleration DS	12378	33	616	141
Total	18522	47	1006	201
Facade3				
Heightfield	768	3	72	9
Material	768	3	72	9
Acceleration DS	3072	7	252	36
Total	4608	13	1208	140
Facade4				
Heightfield	1059	4	73	4
Material	1059	4	73	4
Acceleration DS	4236	9	90	78
Total	6354	17	236	86

We compare kilobytes of storage for four representations: NOCOMP—no compression, RAR—rar compressed format, SVD—SVD-based factorization, and BIN—binary factorization.

7 Results

The results are structured in three parts. First, we compare the compression rates for selected texture maps to the state of the art. Second, we compare rendering times for a single building to previous work in displacement mapping. Third, we compare rendering times and memory consumption for a fly over of a larger city to triangle-based rendering. The test computer uses a GeForce 8800 GTX graphics card.

Compression rates. We modeled four facades to evaluate compression rates (see Table 1). Each facade consists of a displacement map, a material index map, an acceleration data structure map, and an indexed acceleration data structure map. We compare several different algorithms:

1. No compression (NOCOMP).
2. WinRar, a software tool that provides lossless compression based on LZSS and Huffman encoding. This algorithm gives a reasonable estimate for an upper bound of compression ratios we can achieve. WinRar is a significantly more complex algorithm that would take several orders of magnitude longer to decompress.
3. Factorization based on the SVD (SVD).
4. Binary factorization (BIN).

Fig. 12. This figure shows the four test facades used in the result section of the paper.

The facades we selected for the tests are shown in Fig. 12. The image sizes of the facades are F1: 600 × 1,000, F2: 1,536 × 2,048, F3: 1,024 × 768, and F4: 930 × 1,166. We computed that the facades would require between 1,000 to 5,000 triangles in a triangular representation. Please note that our new algorithm, binary factorization, results in representations that are up to 18 times smaller than compressed representations computed with SVD-based factorization. Please also note that most previous graphics algorithms used factorization for lossy compression. As a result, they would typically only keep the most important four to eight outer products. As our algorithm uses lossless compression, we have to expect to keep more values. In our tests, we needed 12-24 outer products to compress the displacement maps using SVD and 32-96 for the binary factorization. To compress the acceleration data structures, we keep up to 44 outer products for SVD-based factorization and up to 480 outer products using binary factorization.

TABLE 2
Rendering Speeds for Four Selected Facades

	Rendering Method	F1	F2	F3	F4
uncompressed displacement mapping					
	RM (previous work)	443	94	360	209
	FDM	730	387	712	369
compressed displacement mapping					
	CRM (previous work)	30	28	34	35
	BCRM	110	67	111	43
	CFDM	219	82	137	116
	BCFDM	213	124	230	104

We compare average frames per second for several rendering algorithms: RM—Relief Mapping without compression (20 linear and 6 binary steps), FDM—Facade Displacement Mapping without compression, CRM—Relief mapping with compression using SVD (20 linear and 6 binary steps), BCRM—Relief Mapping with compression using BMF, CFDM—facade displacement mapping with compression using SVD, and BCFDM—facade displacement mapping with compression using BMF.

The compression algorithm works better for the height field and material information than for the acceleration data structure. The acceleration data structure contains a type of coherence that is not a very good match for the binary factorization. For example, consider a simple model with rectangular facade elements having identical displacements (see Fig. 8). The displacements and material information can be encoded in a single term using binary factorization. However, the values in the acceleration data structure have only some coherence in the vertical and horizontal direction so that multiple terms are needed.

Rendering speed. We compare the rendering speed for the four selected facades by recording short animation sequences of 1,000 frames and measuring the frames per second. We render using 1,600 $\times$ 1,200 screen resolution. We compare several different algorithms. The first two algorithms are uncompressed displacement mapping strategies, and algorithms 3-6 are compressed rendering algorithms.

1. Policarpo's Relief Mapping method (RM) using modified code from his web page [19] with 20 linear and 6 binary steps,
2. Facade Displacement Maps without compression (FDM),
3. Policarpo's relief mapping method using the SVD compression algorithm (CRM),
4. Policarpo's relief mapping method using our BMF compression algorithm (BCRM),
5. Compressed Facade Displacement Maps algorithm with SVD compression (CFDM), and
6. Binary Compressed Facade Displacement Maps algorithm using our BMF compression (BCFDM).

See Table 2 for an overview of the results. Our main observation is the following: our uncompressed displacement mapping algorithm (FDM) gives speedup factors of 1.6, 4.1, 2, and 1.8 compared to previous work (RM). For compressed displacement mapping, our best algorithm (BCFDM) gives speedup factors of 7.1, 4.4, 6.8, and 3.0 compared against CRM. We also want to make two remarks. First, our algorithm is not only faster, but it is also necessary to achieve good visual quality. Second, we consider relief mapping the state of the art, even though there are several other algorithms that perform faster in the uncompressed version. However, as mentioned in Section 4, all other algorithms require data structures that do not compress well and therefore lose their advantage over relief mapping in compressed rendering.

Scalability. We created a large city to test the scalability of our algorithm and to compare it to geometry-based rendering. See Fig. 13. The city consists of 30,000 buildings. Each building has four or more facades. We recorded a fly over of the city and compared two rendering methods. The first method is BCFDM, our best compressed displacement mapping technique. The second method is geometry rendering using textured triangles as a primitive. For polygonal rendering, we triangulated the facades to give the same geometric detail as a displacement mapper (about 1,500 triangles per facade), and we use a 512 $\times$ 512 texture for appearance. We used instancing for rendering, because otherwise, the comparison would be impossible. See Table 3 for the results.

We want to emphasize a few observations. To accelerate displacement mapping using early z-culling, the scene can be rendered once to initialize the z-buffer, or it can be traversed from front to back. As expected, our algorithm is mainly fill limited, while the geometry-based rendering is transform limited. We see that the speedup that our method can achieve ranges from a factor of 40 to 80, depending on the resolution. Additionally, we can observe that geometry-based rendering requires a factor of 14.5 more memory. It is not possible to efficiently use hardware-based occlusion queries to cull single buildings, because we were recording a fly over and not a walk-through. We used a simple city for this test to avoid overly complicated state switching and texture sorting for the geometry-based renderer. The simple city can be rendered as geometry by instancing several large vertex buffer objects. The city uses six different facade detail maps.

8 Discussion

Importance. We believe that the current strong interest in urban reconstruction will result in detailed city models being generated over the next three to five years. The vast amount of generated data will be difficult to manage, and therefore, it seems to be important to explore compressed representations that can be easily integrated into the rendering pipeline. This will allow us to create better fly overs and walk-throughs, because memory management has become a problematic bottleneck in real-time rendering. Our approach is a modern rendering strategy, because we render coarse geometry as triangles and add details by fragment shader ray tracing [52]. While the approach does not provide an advantage over triangle-based rendering for single buildings, we demonstrated that our rendering speeds are much faster for larger environments. This paper is a first step in the direction of investigating compressed representations for massive urban rendering, and we hope that our promising results invite others to join this research direction.

Limitations. The first limitation of the compression and rendering algorithm is that it only works with matrices (textures) that can be partitioned into larger submatrices with identical entries. We believe that this is a reasonable assumption for most building facades. We argue that this

Fig. 13. A complete city rendered with Compressed Facade Displacement Maps. The city consists of 30,000 buildings.

class of matrices is important enough so that it warrants a separate solution for rendering and compression. In future work, we plan to address the compression of general displacement maps to complement our compression algorithm for facade displacement maps. We intend to use constrained matrix factorization algorithms to cull the computation of outer products.

Comparison to other compression algorithms. It is difficult to make a formal comparison of compression algorithms, because we focus on a special class of matrices (textures), and we require very fast decompression. We did not expect to outperform strong algorithms such as rar with our compression technique, but we were surprised to see a significant improvement over SVD. We believe that matrix factorization is a great tool for compression in the context of real-time rendering, because it allows random access by decompressing any entry in the matrix separately. During the course of this research project, we also experimented with other techniques such as the wavelet transform and tensor factorization, but the decompression is much more sophisticated and cannot easily be integrated into a shader program.

9 Conclusions

This paper presents a method to efficiently render massive urban models. Our main idea is to model the city using displacement maps and to render from a compressed representation directly. We demonstrated that our rendering framework outperforms existing displacement mapping techniques and geometry-based rendering for large models. We believe that texture compression will be a crucial aspect of the future development of real-time rendering, and we are interested to pursue related avenues of research.

Appendix

Source Code

We briefly discuss the implementation of the reconstruction of binary factorized matrices. Let us look at the code Listing 1. We take as input three sampler variables containing the X, Y, and S matrices, input texture coordinates, and the number of iterations required for reconstruction. We start a loop that performs the reconstruction. There are a couple of important things to note. Recall that when the original matrices are encoded in textures every 32-bit texel contains 32 elements of the original matrix. Thus, in one lookup, we have access to 32 matrix elements. For us, the value of the integer looked up from the texture has no meaning; we treat it like a binary

TABLE 3
This Table Compares the Rendering Speed and Memory Consumption of Our Algorithm against Geometry-Based Rendering

	Our Algorithm	Geometry
1600 × 1200 screen size	21.5 fps	0.5 fps
1200 × 900 screen size	30.2 fps	0.6 fps
800 × 600 screen size	46 fps	0.6 fps
Uncompressed Size in GB	138	23.93
Compressed Size in GB	1.62	N/A

string because we are interested in individual bits. The simple way to reconstruct would be to knock out 1 bit at a time using bitwise mask and multiply. This has complexity $O(n)$, where n is the length of the binary string. We observe that the matrices obtained by BMF are sparse, and most of the elements are zeros. These do not contribute to the reconstruction, and it is inefficient to perform bitwise shifts and multiplications for each zero encountered. We use a method based on discrete logarithms [53] to efficiently find the rightmost one in a binary string. Inside the loop, we find the position of the rightmost one and right shift the string so that this one is shifted out. We then look up the matrix S in the appropriately shifted location and simply add the looked up value to our partial sum. The loop terminates when all the ones have been shifted out at which point the value of the integer becomes zero.

Listing 1 bmflookup_grayscale: Lookup routine for grayscale images factorized with BMF

```
float bmflookup_grayscale(
    uniform isamplerRECT X,
      uniform samplerRECT S,
      uniform isamplerRECT Y,
      float2 texCoords,
      int recon)
{
 float4 sk = float4(0,0,0,0);
 float c = 0.0;
 int4 xk, yk, x_vec, y_vec;
 int xk_and_yk = 0, position = 0, i = 0;

 for (int k = 0; k < recon; k += 32)
 {
  xk = texRECT(X,
   float2 (float(k/32), texCoords.y));
  yk = texRECT(Y,
   float2(float(k/32),texCoords.x));
  xk_and_yk = xk.x & yk.x;

  for (int j = 0; xk_and_yk != 0; j ++)
  {
   position = discrete_logs [
    (xk_and_yk & (-xk_and_yk)) % 37];
   xk_and_yk = xk_and_yk >>
    (position + 1);
   i = i + position + sign(j);
   sk = texRECT(S, float2(k + i, 0.0)) . x;
   c += sk.x;
  }
 }
 return c;
}
```

ACKNOWLEDGMENTS

The authors thank Pushpak Karnick for code reviews and facade modeling, John Owens for paper review and feedback, and all reviewers for their helpful comments. The authors would like to acknowledge the support of US National Science Foundation (NSF) Contracts IIS 0612269, CCF 0811790, and IIS 0757623 and NGA Contract HM1582-05-1-2004.

REFERENCES

[1] D.W. Paglieroni and S.M. Petersen, "Height Distributional Distance Transform Methods for Height Field Ray Tracing," *ACM Trans. Graphics,* vol. 13, no. 4, pp. 376-399, 1994.

[2] W. Donnelly, "Per-Pixel Displacement Mapping with Distance Functions," *GPU Gems 2,* 2005.

[3] J. Kautz and M.D. McCool, "Interactive Rendering with Arbitrary BRDFs Using Separable Approximations," *Proc. Eurographics Workshop Rendering Techniques,* 253-253, 1999.

[4] L. Borgeat, G. Godin, F. Blais, P. Massicotte, and C. Lahanier, "GoLD: Interactive Display of Huge Colored and Textured Models," *ACM Trans. Graphics,* vol. 26, no. 3, pp. 869-877, July 2005.

[5] E. Gobbetti and F. Marton, "Far Voxels: A Multiresolution Framework for interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms," *ACM Trans. Graphics,* vol. 24, no. 3, pp. 878-885, July 2005.

[6] P. Lindstrom, D. Koller, W. Ribarsky, L.F. Hodges, N. Faust, and G.A. Turner, "Real-Time, Continuous Level of Detail Rendering of Height Fields," *Proc. ACM SIGGRAPH '96,* vol. 30, no. Ann. Conf. Series, pp. 109-118, 1996.

[7] F. Losasso and H. Hoppe, "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids," *ACM Trans. Graphics,* vol. 23, no. 3, pp. 769-776, Aug. 2004.

[8] 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,* vol. 25, no. 3, Sept. 2006.

[9] F. Musgrave, "Grid Tracing: Fast Ray Tracing for Height Fields," technical report, 1988.

[10] D.W. Paglieroni, "The Directional Parameter Plane Transform of a Height Field," *ACM Trans. Graphics,* vol. 17, no. 1, pp. 50-70, 1998.

[11] C.-H. Lee and Y.-G. Shin, "A Terrain Rendering Method Using Vertical Ray Coherence," *J. Visualization and Computer Animation,* vol. 8, no. 2, pp. 97-114, 1997.

[12] B.E. Smits, P. Shirley, and M.M. Stark, "Direct Ray Tracing of Displacement Mapped Triangles," *Proc. Eurographics Workshop Rendering Techniques,* pp. 307-318, 2000.

[13] R.L. Cook, "Shade Trees," *Proc. ACM SIGGRAPH '84,* pp. 223-231, 1984.

[14] J.F. Blinn, "Simulation of Wrinkled Surfaces," *Proc. ACM SIGGRAPH '78,* pp. 286-292, 1978.

[15] S. Gumhold and T. Huettner, "Multiresolution Rendering with Displacement Mapping," *Proc. ACM SIGGRAPH/Eurographics Workshop Graphics Hardware (HWWS '99),* pp. 55-66, 1999.

[16] M. Doggett and J. Hirche, "Adaptive View Dependent Tessellation of Displacement Maps," *Proc. ACM SIGGRAPH/Eurographics Workshop Graphics Hardware (HWWS '00),* pp. 59-66, 2000.

[17] K. Moule and M.D. McCool, "Efficient Bounded Adaptive Tessellation of Displacement Maps," *Proc. Conf. Graphics Interface (GI '02),* pp. 171-180, May 2002.

[18] J. Hirche, A. Ehlert, S. Guthe, and M. Doggett, "Hardware Accelerated Per-Pixel Displacement Mapping," *Proc. Conf. Graphics Interface (GI '04),* pp. 153-158, 2004.

[19] F. Policarpo, M.M. Oliveira, and A.L.D.C. Jo, "Real-Time Relief Mapping on Arbitrary Polygonal Surfaces," *Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics and Games (I3D '05),* pp. 155-162, 2005.

[20] E.A. Risser, M.A. Shah, and S. Pattanaik, "Interval Mapping," technical report, School of Eng. and Computer Science, Univ. of Central Florida, 2006.

[21] L. Baboud and X. Décoret, "Rendering Geometry with Relief Textures," *Proc. Conf. Graphics Interface (GI '06),* pp. 195-201, 2006.

[22] J. Dummer, "Cone Step Mapping: An Iterative Ray-Heightfield Intersection Algorithm," technical report, http://www.lonesock.net/files/ConeStepMapping.pdf, 2006.

[23] L. Wang, X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, and H.-Y. Shum, "View-Dependent Displacement Mapping," *ACM Trans. Graphics,* vol. 22, no. 3, pp. 334-339, 2003.

[24] N. Tatarchuk, "Dynamic Parallax Occlusion Mapping with Approximate Soft Shadows," *Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics and Games (I3D '06),* pp. 63-69, 2006.

[25] G.H. Golub and C.F. Van Loan, *Matrix Computations,* third ed. Johns Hopkins Univ. Press, 1996.

[26] I.T. Jolliffe, *Principal Component Analysis.* Springer, 1986.

[27] A. Fournier, "Separating Reflection Functions for Linear Radiosity," *Proc. Eurographics Workshop Rendering Techniques,* pp. 296-305, 1995.
[28] R. Wang, J. Tran, and D.P. Luebke, "All-Frequency Relighting of Non-Diffuse Objects Using Separable BRDF Approximation," *Proc. 15th Eurographics Workshop Rendering Techniques,* pp. 345-354, 2004.
[29] M. Alexa and W. Müller, "Representing Animations by Principal Components," *Computer Graphics Forum,* vol. 19, no. 3, 2000.
[30] K. Nishino, Y. Sato, and K. Ikeuchi, "Eigen-Texture Method: Appearance Compression Based on 3D Model," *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR '99),* pp. 618-624, 1999.
[31] M. Turk and A. Pentland, "Eigenfaces for Recognition," *J. Cognitive Neuroscience,* vol. 3, no. 1, pp. 71-86, 1991.
[32] J. Lehtinen and J. Kautz, "Matrix Radiance Transfer," *Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics (I3D '03),* pp. 59-64, Apr. 2003.
[33] L. Latta and A. Kolb, "Homomorphic Factorization of BRDF-Based Lighting Computation," *Proc. ACM SIGGRAPH '02,* pp. 509-516, 2002.
[34] M.D. McCool, J. Ang, and A. Ahmad, "Homomorphic Factorization of BRDFs for High-Performance Rendering," *Proc. ACM SIGGRAPH '01,* pp. 171-178, 2001.
[35] F. Suykens, K. vom Berge, A. Lagae, and P. Dutré, "Interactive Rendering with Bidirectional Texture Functions," *Computer Graphics Forum,* vol. 22, no. 3, pp. 463-472, Sept. 2003.
[36] D.D. Lee and H.S. Seung, *Algorithms for Non-Negative Matrix Factorization.* MIT Press, pp. 556-562, 2000.
[37] J. Lawrence, S. Rusinkiewicz, and R. Ramamoorthi, "Efficient BRDF Importance Sampling Using a Factored Representation," *ACM Trans. Graphics,* vol. 23, no. 3, pp. 496-505, 2004.
[38] J. Lawrence, A. Ben-Artzi, C. DeCoro, W. Matusik, H. Pfister, R. Ramamoorthi, and S. Rusinkiewicz, "Inverse Shade Trees for Non-Parametric Material Representation and Editing," *ACM Trans. Graphics,* vol. 25, no. 3, pp. 735-745, 2006.
[39] P. Peers, K. vom Berge, W. Matusik, R. Ramamoorthi, J. Lawrence, S. Rusinkiewicz, and P. Dutré, "A Compact Factored Representation of Heterogeneous Subsurface Scattering," *ACM Trans. Graphics,* vol. 25, no. 3, pp. 746-753, 2006.
[40] P.-P.J. Sloan, J. Hall, J.C. Hart, and J. Snyder, "Clustered Principal Components for Precomputed Radiance Transfer," *ACM Trans. Graphics,* vol. 22, no. 3, pp. 382-391, 2003.
[41] A.W. Kristensen, T. Akenine-Möller, and H.W. Jensen, "Precomputed Local Radiance Transfer for Real-Time Lighting Design," *ACM Trans. Graphics,* vol. 24, no. 3, pp. 1208-1215, 2005.
[42] M. Sattler, R. Sarlette, and R. Klein, "Simple and Efficient Compression of Animation Sequences," *Proc. ACM SIGGRAPH/ Eurographics Symp. Computer Animation (SCA '05),* pp. 209-218, 2005.
[43] O. Arikan, "Compression of Motion Capture Databases," *ACM Trans. Graphics,* vol. 25, no. 3, pp. 890-897, http://doi.acm.org/10.1145/1141911.1141971, 2006.
[44] J. Ricard, J. Royan, and O. Aubault, "From Photographs to Procedural Facade Models," *ACM SIGGRAPH '07,* p. 75, 2007.
[45] P. Müller, G. Zeng, P. Wonka, and L.V. Gool, "Image-Based Procedural Modeling of Facades," *ACM Trans. Graphics,* vol. 24, no. 3, p. 85, 2007.
[46] G. Schaufler, J. Dorsey, X. Decoret, and F. Sillion, "Conservative Volumetric Visibility with Occluder Fusion," *Proc. ACM SIGGRAPH '00,* pp. 229-238, 2000.
[47] Y.I.H. Parish and P. Müller, "Procedural Modeling of Cities," *Proc. ACM SIGGRAPH '01,* E. Fiume, ed., pp. 301-308, 2001.
[48] W.H. Press, W.T. Vetterling, S.A. Teukolsky, and B.P. Flannery, *Numerical Recipes in C++: The Art of Scientific Computing,* 2002.
[49] M. Koyutürk and A. Grama, "PROXIMUS: A Framework for Analyzing Very High Dimensional Discrete-Attributed Datasets," *Proc. ACM SIGKDD '03,* pp. 147-156, 2003.
[50] T. Kolda and D. O'Leary, "A Semidiscrete Matrix Decomposition for Latent Semantic Indexing Information Retrieval," *ACM Trans. Information Systems,* vol. 16, no. 4, pp. 322-346, 1998.
[51] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.
[52] D. Blythe, "The Direct3d 10 System," *ACM Trans. Graphics,* vol. 25, no. 3, pp. 724-734, 2006.
[53] C. Sturtivant, *Finding the Right One,* http://blog.lib.umn.edu/sturt001/sturtivant/2006/12/finding_the_right_one.html, 2008.

Saif Ali received the BS degree in computer engineering from Jamia Millia Islamia, New Delhi, India, and the MS degree in computer science and a concentration in arts, media, and engineering from Arizona State University in September 2007. He is a member of the Stream Computing SDK team at AMD, where he writes code for GPGPU on next-generation graphics hardware. He cultivates a parallel interest in photography.

Jieping Ye received the PhD degree in computer science from the University of Minnesota-Twin Cities in 2005. He is currently an assistant professor in the Department of Computer Science and Engineering, Arizona State University. He has been a core faculty member of the Center for Evolutionary Functional Genomics, Bio-design Institute, Arizona State University, since August 2005. His research interests include machine learning, data mining, and bioinformatics. He has published extensively in these areas. He received the Guidant Fellowship in 2004 and 2005. In 2004, his paper on generalized low-rank approximations of matrices won the outstanding student paper award at the 21st International Conference on Machine Learning. He is a member of the IEEE and the ACM.

Anshuman Razdan received the BS and MS degrees in mechanical engineering and the PhD degree in computer science. He is an associate professor in the Division of Computing Studies and the Director of Advanced Technology Innovation Collaboratory (ATIC) and the I3DEA Laboratory (i3deas.asu.edu) at Arizona State University (ASU), Polytechnic campus. He has been a pioneer in computing-based interdisciplinary collaboration and research at ASU. His research interests include geometric design, computer graphics, document exploitation, and geospatial visualization and analysis. He is the principal investigator and a collaborator on several federal grants from agencies, including the US National Science Foundation (NSF), NGA, and NIH. He is a member of the IEEE.

Peter Wonka received the MS degree in urban planning and the doctorate in computer science from the Technical University of Vienna. He is currently with Arizona State University (ASU). Prior to coming to ASU, he was a postdoctorate researcher at the Georgia Institute of Technology for two years. His research interests include various topics in computer graphics, visualization, and image processing. He is a member of the IEEE.