

Adaptive Texture Maps

Martin Kraus and Thomas Ertl[†]

Visualization and Interactive Systems Group, Universität Stuttgart, Germany

Abstract

We introduce several new variants of hardware-based adaptive texture maps and present applications in two, three, and four dimensions. In particular, we discuss representations of images and volumes with locally adaptive resolution, lossless compression of light fields, and vector quantization of volume data. All corresponding texture decoders were successfully integrated into the programmable texturing pipeline of commercial off-the-shelf graphics hardware.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism; I.4.2 [Image Processing and Computer Vision]: Compression (Coding); I.4.10 [Image Processing and Computer Vision]: Image Representation

1. Introduction

Although texture mapping is one of the most important and best researched techniques in computer graphics, it is still posing many challenges, in particular the problem of storing and accessing large texture maps with fine details. On the other hand, the availability of programmable per-pixel operations (including dependent texture mapping) in low-cost graphics hardware is likely to provide new, attractive solutions for many fundamental tasks in real-time rendering.

Specifically, today's programmable graphics hardware allows us to integrate decoders for texture data in the rasterization pipeline. Thus, only compressed data needs to be stored in dedicated texture memory provided that each texture lookup includes a decoding step, which is, however, performed within the hardware's rasterization pipeline. Not only is the texture decoding in hardware considerably faster than comparable software decoding, we also avoid bandwidth limitations as the compressed texture data may often reside permanently in the texture memory while software decoders have to send the decompressed data over the graphics bus to the texture memory.

Apart from the obligatory need for fast random access, there are — depending on the particular application — several different, partially contradicting requirements for hardware implementations of texture mapping. Our primary goal, however, is to design adaptive texture maps, i.e. texture maps with locally adaptive resolution and an adaptive boundary of the texture map's domain instead of a rectilinear domain. Benefits of this two-fold adaptivity are well-known from unstructured meshes, e.g. triangle meshes. For example, adaptive boundaries save considerable memory costs if the texture data contains large empty regions often encountered in billboards, volume data, and light fields. Moreover, the locally adaptive resolution permits continuous simplification, i.e. lossy compression, and allows us to efficiently represent fine details without the need to increase the resolution (and therefore size) of the whole texture map.

We summarize previous work about adaptive texture mapping and texture decoding in graphics hardware in Section 2. In Section 3, our basic design for adaptive texture maps is introduced and the basic features of our approach are illustrated with the help of a two-dimensional texture image. Variants and applications of adaptive texture mapping in three and four dimensions are presented in Section 4. Specifically, we discuss a three-dimensional variant for volume rendering and four-dimensional texture maps with adaptive boundaries for light field rendering. As our approach to adaptive texture mapping includes data com-

[†] IfI, Universität Stuttgart, Breitwiesenstr. 20-22, 70565 Stuttgart, Germany. E-mail: {Martin.Kraus | Thomas.Ertl}@informatik.uni-stuttgart.de.

pression with vector quantization as a special case, we also present a hardware-accelerated decoder for vector-quantized volume data. In Section 5, we point out directions for future work and present our conclusions.

2. Related Work

There is a considerable amount of research on simplification and compression of two-dimensional texture maps for triangulated surfaces that relies on the degrees of freedom offered by texture coordinates, which are specified at each vertex of a triangular mesh. In these works, the texture mapping is (implicitly) interpreted as a composite mapping from geometric coordinates to texture coordinates and then from texture coordinates to texture data, e.g. color. As we are concerned with texture mapping in the more restricted sense of mapping texture coordinates to texture data, we only mention the work by Yu et al.¹⁶ since they employed ideas from vector quantization (see Section 4.2 and Gersho and Gray⁶) in order to decide how to reuse triangular texture patches. They encountered serious artifacts at the boundaries of texture patches, which are also a major concern in this work. However, our solution to this problem is almost orthogonal and the remaining artifacts are of a completely different nature.

More related to our concept of adaptive texture mapping are the previously published variants of adaptive texture maps, e.g. Heckbert⁸, or Fernando et al.⁴. However, these concepts are usually based on hierarchical data structures and, therefore, rather inappropriate for an efficient implementation in today's graphics hardware. Hardware concepts for compressed texture maps have been proposed, e.g. for the Talisman architecture¹⁴, and implemented, e.g. vector quantization in the PowerVR architecture and the S3 texture compression¹³, which is a de facto standard in today's programmable graphics hardware. Unfortunately, the accuracy of the S3 texture compression is often insufficient and the scheme is hardly customizable.

Although implemented in software only and limited to vector quantization, several previously proposed data compression schemes^{2, 10, 11, 12} are strongly related to our work, while wavelet compression schemes, e.g. Bajaj et al.¹, are usually considerably more sophisticated and, therefore, more difficult to implement in graphics hardware. Vector quantization was proposed for the compression of texture images by Beers et al.², for four-dimensional light fields by Levoy and Hanrahan¹⁰, and for volume data by Ning and Hesselink^{11, 12}. Although volume textures were not mentioned explicitly by Ning and Hesselink^{11, 12}, the proposed compression scheme is also applicable to texture compression, provided the volume rendering algorithm is replaced by an appropriate method, e.g. the algorithm proposed by Cabral et al.³, which is briefly explained in Section 4.1.

3. Adaptive Texture Mapping

As mentioned in the introduction, we present our approach to adaptive texture mapping for two-dimensional texture maps in this section and extend it to three and four dimensions in Section 4. We emphasize that the example presented in this section is for the purpose of illustration only. In fact, our approach appears to be more useful in three and four than in two dimensions; however, an explanation of the two-dimensional case is likely to be more comprehensible than a discussion of the general case.

This section begins with a specification of our requirements, which are in fact restrictive enough to determine our basic data structures, which in turn lead to the decoding scheme, i.e. the algorithm for a texture lookup. Furthermore, we present our implementation of the decoder in current programmable graphics hardware, namely the ATI Radeon 8500, and discuss the generation of adaptive texture maps.

3.1. Requirements

Our primary goal is an adaptive representation of texture data that features a locally varying resolution and an adaptive boundary of the texture map's domain. Moreover, we intend to implement the decoder within the rasterization pipeline of off-the-shelf programmable graphics hardware, such that the texture data is decoded "on the fly" for each texture lookup. This approach allows for the fast, random access, which is usually required for texture decoding techniques^{1, 2, 10, 11}.

As the flexibility of current programmable graphics hardware, e.g. the ATI Radeon 8500 or NVIDIA's GeForce3, is rather limited, we restrict ourselves to only one level of indirection, which is implemented with dependent texture mapping. Therefore, our data structures may include references but no nested references. Furthermore, we require a continuous interpolation of the texture data, i.e. a continuous texture map. However, we relax this requirement in Section 4.2, as we could not implement a decoder for vector-quantized data in current graphics hardware otherwise.

3.2. Representation of Adaptive Texture Maps

In the context of meshing, there are basically two approaches to locally adaptive resolutions: hierarchical meshes and unstructured meshes. The latter are not suitable in our case, because of the problem of cell location, which is part of any random access in an unstructured mesh. On the other hand, hierarchical meshes are also inappropriate, as any hierarchy implies the need for nested references, which we have to avoid. Our solution is to employ a hierarchical representation that is restricted to just two levels. Thus, we retain a restricted form of locally adaptive resolution and at the same time avoid the nesting of references.

The first (upper) level of our representation is a coarse, uniform grid covering the domain of the texture map. The

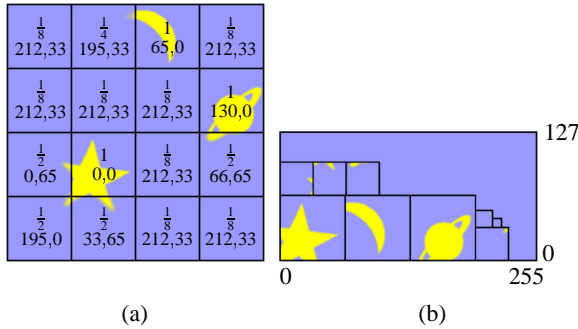


Figure 1: Representation of adaptive texture maps. (a) Index data: scale factors and coordinates of packed data blocks are stored for each cell of a 4×4 grid representing the whole texture map, which is included for the purpose of illustration only. Actual coordinates are between 0 and 1. (b) Packed data: the data blocks packed into a uniform grid of 256×128 texels. The blocks' frames are for illustration purposes only.

data defined on this grid will be called the *index data*. For each cell, these data consist of one reference to the texture data of the cell, which is called a *data block*, and a scaling factor specifying its resolution relatively to the maximum resolution; an example is given in Figure 1a. The second (lower) level contains the actual data blocks remapped to a uniform resolution such that all data blocks may be packed into one uniform grid; therefore, these data will be called the *packed data*; see Figure 1b for an example. Although the cells of the coarse grid are of uniform size, the packed data blocks are of different sizes depending on their resolution, i.e. data blocks of a high resolution will correspond to large blocks of the packed data because of the remapping to a uniform resolution.

In order to guarantee continuous texture mapping, i.e. a continuous interpolation of the texture data, we replicate the texels of the data blocks' boundaries and employ bilinear interpolation of the texels' data. This corresponds to the *space-filling block arrangement* suggested by Ning and Hesselink¹² and is illustrated in Figure 2. Note that (in contrast to the OpenGL definition of textures) texel values are specified at vertices and the domain of valid texture coordinates is limited by the vertices' positions in order to allow for the bilinear interpolation. For a block of size $b \times b$ texels, the replication of boundary texels causes an increase in the amount of data by a factor of about $(b+1)^2/b^2$, e.g. an acceptable memory overhead of 13% for image blocks of size 16×16 texels.

Neither our requirements nor the chosen data structures impose any restriction on multiple references to a single data block. In particular, we propose to always include an *empty data block*, which is referenced (at least) by all cells of the coarse grid outside the domain of the texture map. All texels of the empty data block are set to an *empty texel value*,

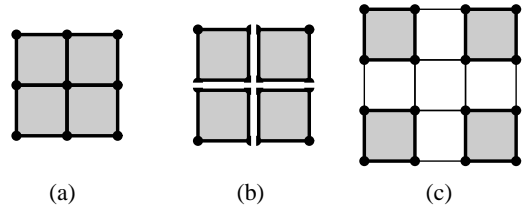


Figure 2: Decomposition of a 3×3 grid into four blocks. Bilinear interpolation may be employed within the gray regions. (a) The original grid. (b) Decomposition into four 2×2 blocks. Vertices at the blocks' boundaries are replicated. (c) Packing of the four blocks into one 4×4 grid.

which depends on the particular application. For color images this value is usually the background's color or a completely transparent color. As the empty data block is perfectly homogeneous, it may be stored with the minimum resolution, i.e. the minimum block size.

Before presenting one particular way of computing the index data and packed data in Section 3.5, we discuss the sampling of adaptive texture maps and its implementation in the next two sections since the efficient sampling of adaptive texture maps is the most important requirement for the chosen representation.

3.3. Sampling of Adaptive Texture Maps

A texture lookup in an adaptive texture map at texture coordinates (s, t) (see also Figure 3) is performed in several steps, which are:

1. determination of the cell of the index data that includes the point (s, t) ;
2. computation of the coordinates (s_o, t_o) corresponding to the origin of this cell;
3. lookup of the index data for this cell, i.e. of the associated scale factor m and the origin (s'_o, t'_o) of the data block in the packed data;
4. computation of the coordinates (s', t') in the packed data corresponding to (s, t) in the index data; and
5. lookup and interpolation of the actual texture data at (s', t') in the packed data.

The dimensions of the index data are denoted by n_s and n_t , i.e. there are $n_s \times n_t$ cells in the coarse grid of the upper level of the adaptive texture map's representation. n'_s and n'_t are the dimensions of the packed data, i.e. all data blocks are packed into a grid of $n'_s \times n'_t$ texels. Additionally, we define the maximum resolution of a data block by a maximum size of $b_s \times b_t$ texels. Note that replicated texels are included in b_s and b_t , e.g. the data blocks of Figure 2c are of size 2×2 although the area for interpolation is only of the size of one texel, or in general of $(b_s - 1) \times (b_t - 1)$ texels. The scale factor m will be set to 1 for this maximum resolution.

These definitions are particularly convenient when the adaptive texture map is derived from a uniform texture image of size $(n_s(b_s - 1)) \times (n_t(b_t - 1))$ because in this case the maximum size of a data block is limited to $b_s \times b_t$.

With these definitions we may compute the origin (s_o, t_o) of the cell including the point (s, t) by

$$s_o = \frac{\lfloor s n_s \rfloor}{n_s} \quad \text{and} \quad t_o = \frac{\lfloor t n_t \rfloor}{n_t}, \quad (1)$$

where the floor function $\lfloor x \rfloor$ gives the largest integer less than or equal to x . The scale factor m and the origin (s'_o, t'_o) of the corresponding packed data block are given as functions of (s_o, t_o) . Thus, we may compute the texture coordinates (s', t') in the packed data by

$$s' = s'_o + (s - s_o) m \frac{n'_s}{n_s(b_s - 1)} \quad \text{and} \quad (2)$$

$$t' = t'_o + (t - t_o) m \frac{n'_t}{n_t(b_t - 1)}, \quad (3)$$

i.e. we scale the offset $(s, t) - (s_o, t_o)$ with m and two additional factors, and add this scaled offset to the origin (s'_o, t'_o) .

These additional factors $n'_s/(n_s(b_s - 1))$ and $n'_t/(n_t(b_t - 1))$ stem from the scaling of all texture coordinates to the range between 0 and 1. In the example depicted in Figure 3, m is 1, n_s and n_t are 4, b_s and b_t are 65, n'_s is 256, and n'_t is 128. Thus, the factor $m n'_s/(n_s(b_s - 1))$ equals 1, but the factor $m (n'_t/(n_t(b_t - 1)))$ equals 1/2. Although the offsets $(s, t) - (s_o, t_o)$ and $(s', t') - (s'_o, t'_o)$ in Figure 3 appear to be equal on first sight, they are actually different as the coordinate system in Figure 3b is only half the height of that in Figure 3a. The factor $n'_t/(n_t(b_t - 1)) = 1/2$ takes care of exactly this difference.

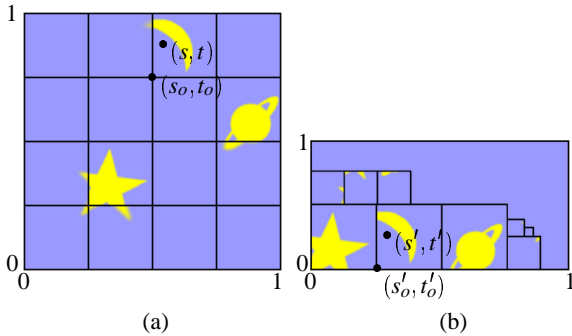


Figure 3: Texture lookup in an adaptive texture map for texture coordinates s and t . (a) (s, t) specifies a cell of the index data, the origin of which is denoted by (s_o, t_o) . (b) (s_o, t_o) corresponds to the origin (s'_o, t'_o) of a packed data block and (s, t) corresponds to a point (s', t') in that data block.

3.4. Implementation of Adaptive Texture Sampling

This section discusses an implementation of the texture lookup described in the previous section on ATI's Radeon

8500 using the "fragment shader" extension⁷. Although the two-dimensional texture lookup could also be implemented on NVIDIA's GeForce3 with the help of the "texture shader" extension⁹, we will not discuss this possibility since the required texture shader programming is less straightforward and we cannot extend it to three or four dimensions.

In our fragment shader implementation, we restrict the grid dimensions n_s , n_t , n'_s , and n'_t to powers of two as these grids are implemented with OpenGL textures. In particular, there is one texture of size $n_s \times n_t$ for the index data and one texture of size $n'_s \times n'_t$ for the packed data. While the latter employs bilinear interpolation and contains the actual image data, the texture for the index data uses nearest-neighbor interpolation and contains for each texel one pair of coordinates (s'_o, t'_o) specifying the origin of the corresponding data block in the texture for the packed data and one scale factor m , i.e. three components per texel, which can be stored in an RGB texture. Note that the limitation to 8 bits of precision for the specification of s'_o and t'_o limits n'_s and n'_t to a maximum value of 256. The scale factor m is restricted to values of the form 2^{-n} with $0 \leq n \leq 7$ as it is also specified by 8 bits. Therefore, the data blocks' dimensions are restricted to values of the form $2^n + 1$ with an integer n greater than or equal to 0, where the term $+1$ stems from the replication of texels at block boundaries.

A fragment shader program on the ATI Radeon 8500 consists of either one or two "passes" each consisting of up to six texture sampling and/or texture coordinate routing instructions followed by up to eight arithmetic instructions. Only the sampling instructions of the second pass may be dependent texture lookups, i.e. their texture coordinates may be the results of previous instructions. This makes a total of four blocks of instructions, which are illustrated in Figure 4. The particular instructions in the four blocks of our fragment shader program are discussed in the next three paragraphs.

The first block of texture sampling instructions of our fragment shader program has to fetch the scale factor m and the coordinates s'_o and t'_o by one nearest-neighbor lookup in the $n_s \times n_t$ index data grid at texture coordinates s and t . Additionally, the coordinates s_o and t_o corresponding to the origin of the fetched texel (see Section 3.3, Equation 1) may be computed from s and t by a second nearest-neighbor texture lookup in another $n_s \times n_t$ texture containing coordinates $(i/n_s, j/n_t)$ in the (i, j) -th texel with $0 \leq i < n_s$ and $0 \leq j < n_t$. Alternatively, s_o and t_o may be computed separately by two texture lookups in two one-dimensional textures containing values i/n_s in the i -th texel with $0 \leq i < n_s$ and j/n_t in the j -th texel with $0 \leq j < n_t$, respectively.

The following block of arithmetic instructions computes the texture coordinates s' and t' for the lookup in the packed data as described in Section 3.3, in particular Equations 2 and 3. Note that the terms $n'_s/(n_s(b_s - 1))$ and $n'_t/(n_t(b_t - 1))$ are constant for all texels; therefore, the computation reduces to one subtraction, two multiplications, and one ad-

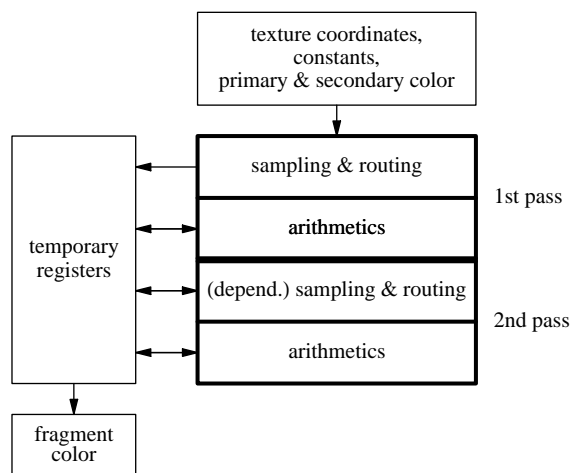


Figure 4: Scheme of the structure of a two-pass fragment shader program, its inputs, temporary registers, and the resulting fragment color.

dition for each coordinate, which can be performed by four fragment shader instructions since these vector instructions may affect up to four components (red, green, blue, and alpha). Moreover, one of the products and the final sum can be evaluated by a single “MAD” (multiply and add) instruction. However, an additional addition is necessary as the center of the (i, j) -th texel in the $n'_s \times n'_t$ texture is at the coordinates $((i + 1/2)/n'_s, (j + 1/2)/n'_t)$; thus, we have to add the constant vector $(1/(2n'_s), 1/(2n'_t))$ to (s', t') . Furthermore, additional operations are necessary in order to reduce artifacts stemming from the limited precision of the fragment shader’s arithmetics. In particular, decreasing m slightly (and at the same time increasing $(1/(2n'_s), 1/(2n'_t))$) helps to reduce artifacts significantly as this allows us to restrict the computed coordinates s' and t' to the correct packed data block. However, these modifications introduce new artifacts by causing additional discontinuities at block boundaries. We expect a strong reduction of these artifacts on future graphics hardware supporting more accurate arithmetics.

The second block of texture sampling instructions employs the coordinates s' and t' for a bilinearly interpolated texture lookup in the texture containing the packed data blocks. This completes the computations of our fragment shader program. Of course, additional texture lookups and arithmetic instructions are possible in the second pass, e.g. in order to blend the resulting color with the primary color and/or with colors resulting from further (standard) texture lookups.

3.5. Generation of Adaptive Texture Maps

In this section, one particular way of generating two-dimensional adaptive texture maps from data defined on uniform grids is discussed. If the data is not specified on a uniform grid but on an unstructured mesh or in parameteric form, it has to be resampled to a uniform grid in order to apply the techniques presented in this section. As the generation of an adaptive texture map will usually be a preprocessing step, we will not discuss any optimizations.

In addition to the nomenclature introduced in Section 3.3, the dimensions of the original uniform grid are denoted by N_s and N_t . If one of these dimensions is not a power of two, it has to be increased to the next greater power of two. In this case, the additional texels should be set to the empty texel value such that the additional empty regions are encoded efficiently.

Our algorithm for generating an adaptive texture map consists of the following steps, which will be commented below (compare also with Figure 1):

1. Build a hierarchy of downsampled versions of the original grid with the grid of the i -th level being of size $2^{-i}N_s \times 2^{-i}N_t$ vertices.
2. Given the maximum data block size $b_s \times b_t$ introduced in Section 3.3, decompose the original grid, i.e. the 0-th level of the hierarchy, into $n_s \times n_t$ cells of size $b_s \times b_t$ using the replication of boundary vertices explained in Section 3.2.
3. For each of the cells of step 2., test whether the data values of the cell are “sufficiently” close to the empty data value. In this case, mark the cell as empty; otherwise, determine an “appropriate” scale factor $m = 2^{-i}$ and copy a corresponding data block of size $(m(b_s - 1) + 1) \times (m(b_t - 1) + 1)$ from the data of the i -th level of the grids’ hierarchy.
4. Build a list of data blocks created in the previous step and append an empty data block, which is referenced by all marked cells.
5. Ensure consistent block boundaries by modifying the data blocks of neighboring cells such that data on shared boundaries is identical. This may be performed, for example, with the method proposed by Westermann et al.¹⁵. However, the empty data block must not be modified.
6. Pack all data blocks into a grid of size $n'_s \times n'_t$, which represents the packed data of the adaptive texture map.
7. Based on the cells’ references to data blocks established in steps 3. and 4., the scale factors of step 3., and the positions of the packed data blocks computed in step 6., assemble the cells’ data in an $n_s \times n_t$ grid, which represents the index data of the adaptive texture map.

The downsampling of step 1. should include some filtering in order to minimize approximation errors; currently we employ a simple averaging of neighboring vertices. The implementation may be simplified by choosing the vertex positions of a downsampled grid from the vertex positions of

the original grid. In this case, it is advantageous to choose dimensions of the form $2^n + 1$ for N_s and N_t . Therefore, the size of the i -th grid should be $(2^{-i}(N_s - 1) + 1) \times (2^{-i}(N_t - 1) + 1)$.

In order to choose appropriate dimensions b_s and b_t of the cells of step 2., several dependencies should be considered: The larger the cells are, the smaller is the amount of index data and the smaller is the memory overhead due to replicated boundary texels. On the other hand, our representation becomes more adaptive with smaller cells, i.e. a larger region of the texture map's domain may be covered by the memory-efficient empty data blocks and the resolution of individual cells may be adapted to local features more efficiently. For this reason, the cells depicted in the illustrative Figures 1a and 3a are far too large. The optimal choice depends not only on the size, shape, and dimensions of the texture map's domain but also on the particular texture data. Therefore, we can do no better than recommend to optimize the cells' dimensions for each particular application or even for each texture map.

In step 3., our current criterion for data values "sufficiently" close to the empty data value is a user-specified limit of the L^∞ -norm (maximum norm) of the difference between the interpolated data within the cell and the empty data value. The "appropriate" scale factor $m = 2^{-i}$ is determined by calculating the L^∞ -norm or the L^2 -norm (depending on the application) of the difference between the interpolated data of the 0-th and the i -th level and choosing the maximum i that results in a norm still smaller than a user-defined limit. Of course, any other criterion may be employed, e.g. a position-dependent metric.

It should be noted that our algorithm does not produce multiple references to data blocks apart from the references to the empty data block. If multiple references to other data blocks were allowed, it would be far more complicated to guarantee identical data on shared cell boundaries, which are required for continuous interpolation. Moreover, this requirement is likely to restrict the use of multiple references to data blocks that feature only constant values; however, these blocks are already efficiently compressed because strongly downsampled versions of them are generated in step 3.

Step 6. requires an approximative solution to a variant of the well-known bin-packing problem. Currently, we employ a simple recursive procedure, which fills a rectilinear empty region with blocks of only one size and calls itself recursively for the remaining empty region, which is decomposed into rectilinear parts. The block size is determined by searching for the largest unpacked block that still fits into the empty region. This simple procedure appears to generate sufficiently good packings.

All steps of this algorithm may be generalized to three and more dimensions without complications. However, the implementation of the texture sampling to more dimensions

is less obvious; therefore, it will be discussed in detail in the next section. Once more, we emphasize that our approach to adaptive texture mapping appears to be less useful for two-dimensional applications; in particular because there are more appropriate methods available, which are based on triangulations of texture images. (See also Section 2.)

4. Variants and Applications

This section presents variants of adaptive texture maps in three and four dimensions; however, instead of focusing on particular implementation details, we want to emphasize the diversity of applications of adaptive texture maps.

4.1. Volume Rendering

Volume data may be rendered with the help of three-dimensional texture mapping by blending a set of view-plane-aligned, textured slices into the frame buffer as suggested by Cabral et al.³ and illustrated in Figure 5. However, one of the main problems of this technique is the large amount of texture memory necessary for standard volume textures. Adaptive texture mapping, on the other hand, compresses the texture data and, therefore, allows us to employ three-dimensional texture mapping for data sets, which have been too large for the texture memory of almost all graphics boards in the past.

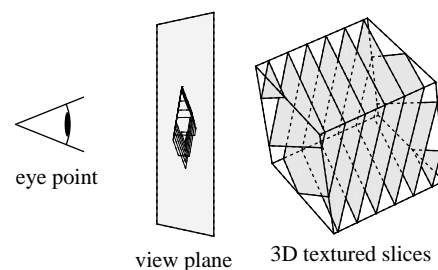


Figure 5: Volume rendering with three-dimensional textures.

The approach discussed in Section 3 is easily generalized to three dimensions, provided that the graphics hardware supports three-dimensional RGBA textures. Moreover, one more texture lookup is required in the first pass of the fragment shader program to determine the three texture coordinates of the origin of the current voxel in the index data (see Section 3.4) since it is usually not practical to employ a (possibly large) three-dimensional texture for this purpose.

In Figures 6c a volume rendering of the $512 \times 512 \times 360 \times 2$ bytes CT scan of the Stanford terra-cotta bunny is depicted. In order to generate the adaptive representation, the data was first converted to a $512^3 \times 1$ byte volume. Then, most of the noise and the pad (see Figure 8) was removed before computing the index and packed data in a variant of the algorithm described in Section 3.5 for three-dimensional data.

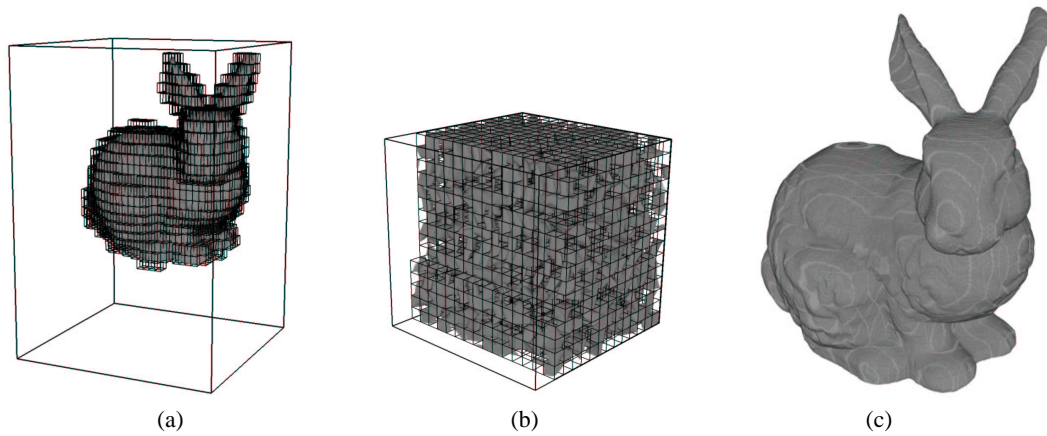


Figure 6: Volume rendering of a $512 \times 512 \times 360$ CT scan with adaptive texture mapping. (a) Non-empty cells of the 32^3 index data grid. (b) Data blocks packed into a 256^3 texture. (c) Resulting volume rendering.

The non-empty cells of the resulting index data, which is stored in a small 32^3 RGBA texture, are shown in Figure 6a. Figure 6b visualizes the packed data blocks, which fit into a 256^3 texture with luminance and alpha components requiring 32 MBytes of texture memory. Our volume renderer achieves a performance of about 6 fps using a 512×512 viewport and about 500 slices on an ATI Radeon 8500 graphics board.

Higher frame rates are possible for smaller viewports, which are, however, not sensible for a volume of virtually 512^3 voxels. As mentioned in Section 3.4, it appears to be impossible to remove all rendering artifacts at block boundaries on current hardware. In Figure 6c, these artifacts manifest themselves as rather large, lighter rings, while the finer ring structures best visible on the forehead and the back of the bunny are due to a sharp transfer function and the limited resolution of the CT scan.

4.2. Vector Quantization of Volume Data

Vector quantization⁶ is an important data compression technique, which was first applied to volume data by Ning and Hesselink^{11, 12}. The basic idea is illustrated in Figure 7: Each cell of the index data specifies one vector of voxels of a codebook. In our application, the codebook includes 256 vectors consisting of 8 bytes corresponding to $2 \times 2 \times 2$ voxels of the CT data. Thus, each cell of the index data specifies the complete data of eight voxels with just one byte, i.e. the compression ratio is about 8 : 1 since the size of the relatively small codebook may be ignored.

This concept for the compression of volume data is similar to our representation of adaptive texture maps provided that we employ nearest-neighbor interpolation instead of a continuous interpolation and set all scale factors to 1, which is in fact a considerable simplification of our approach. In

this case, there is no need to replicate boundary texels and multiple references to one data block are perfectly appropriate. Furthermore, perfect packing of the data blocks, which are all of the same size, becomes trivial; in particular, we may simply build a long row of data blocks and identify each data block by only one coordinate. Thus, the mentioned eight-dimensional vectors correspond to data blocks of $2 \times 2 \times 2$ voxels, the codebook is just the packed data, and the index data consists of only one coordinate per cell, i.e. one byte per cell.

We compressed the $512 \times 512 \times 360 \times 2$ byte CT scan of the terra-cotta bunny to a $256 \times 256 \times 256 \times 1$ byte texture for the index data and generated a codebook of 256 entries with the help of the QccPack software by James Fowler⁵. Then, the codebook was converted to a $2 \times 2 \times 512$ RGBA

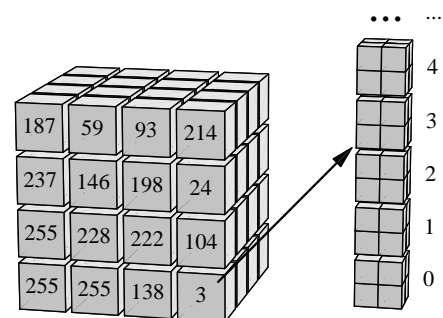


Figure 7: Data structures for vector quantization of volume textures: Each byte of the index data (left) specifies one data block consisting of $2 \times 2 \times 2$ voxels of the packed data (right).

texture by applying transfer functions to the scalar data and a simplified version of our fragment shader program was employed to generate the volume rendering in Figure 8 and Figure 12a on the color page. The frame rate for 500 slices is about 3.5 fps for a 512×512 viewport.

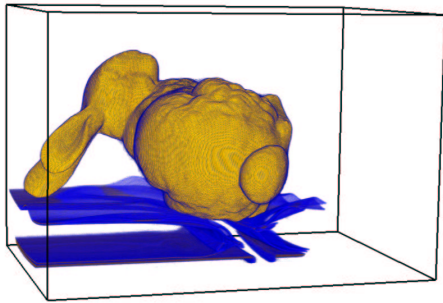


Figure 8: Volume rendering of the vector-quantized volume data of the Stanford terra-cotta bunny.

4.3. Light Field Rendering

Light fields were introduced by Levoy and Hanrahan¹⁰ as four-dimensional functions $L(u, v, s, t)$, which are parameterized by the intersection points of a view ray with a light slab, as illustrated in Figure 9. Approximative rendering of a light field may be performed by rasterizing a textured polygon representing the s - t plane with appropriate texture coordinates u, v, s , and t specified at all vertices. Thus, the problem of rendering a light field is reduced to a four-dimensional texture lookup with quadrilinear interpolation.

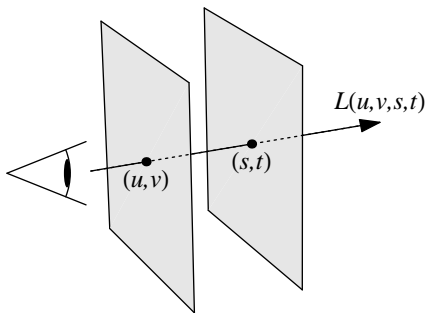


Figure 9: Geometry of a light slab.

Since the ATI Radeon 8500 does not support four-dimensional textures, we have to implement the interpolation by an appropriate weighting of the results of two trilinearly interpolated texture lookups. Moreover, the index data has to be represented by a three-dimensional grid implying that each cell of the index data — and therefore each data block — covers all possible values of one of the coordinates, in our case the v coordinate; see Figure 10.

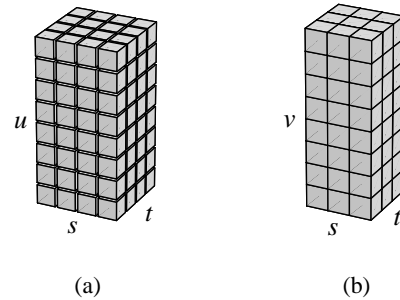


Figure 10: Representation of four-dimensional light fields. (a) Index data grid; each cell covers several values of s and t , one value of u , and all values of v . (b) One of many data blocks; each cell corresponds to a particular quadruplet of values for s, t, u , and v .

On the other hand, each cell covers only one particular value of u , such that two three-dimensional lookups with nearest-neighbor interpolation may be performed in two cells, which are neighbors in the u dimension of the index data. Each of these lookups determines one packed three-dimensional data block and these are used in the second “pass” of our fragment shader program for the two trilinearly interpolated texture lookups. Due to the limited number of arithmetic instructions in the first “pass”, we cannot implement scaling of data blocks; however, all empty cells of the index data are represented by only one data block resulting in a significant compression of the data. Note that applying the techniques of Section 4.2 to light field data would restrict us to an undesirable nearest-neighbor interpolation of colors.

The light field rendered with our approach in Figure 11 and Figure 12b on the color page is a $16 \times 16 \times 256 \times 256$ version of the buddha light field from the Stanford light field

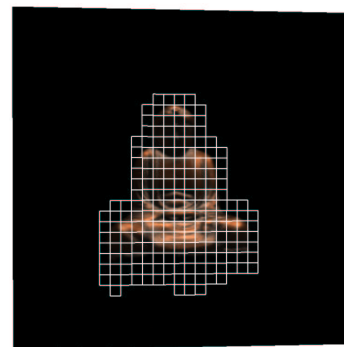


Figure 11: Light field rendering of a $16 \times 16 \times 256 \times 256$ variant of the Stanford buddha light field. Boxes correspond to cell boundaries of non-empty cells in the s and t dimension. (See also Figure 12b on the color page.)

archive, which was decomposed into data blocks of dimensions $16 \times 9 \times 9$ replicating boundary texels only in the s and t dimensions. The non-empty blocks and one empty block were packed into a $128 \times 256 \times 256$ RGB texture occupying 24 MBytes. In Figure 11 boxes indicate the boundaries of non-empty blocks in the s and t dimension. As the reader might expect, slight rendering artifacts are visible at these boundaries in Figure 12b. Since only one textured polygon has to be rasterized in each frame, we achieve a “virtual” frame rate of about 700 fps for a 512×512 viewport without any further optimizations.

5. Future Work and Conclusions

We have demonstrated that today’s programmable graphics hardware allows us to replace the rigid structure of texture maps by an adaptive representation. However, our work also revealed several serious limitations of current hardware, which will hopefully be removed in the near future, e.g. the restriction to one level of dependent texture lookups and the limited precision of per-pixel operations.

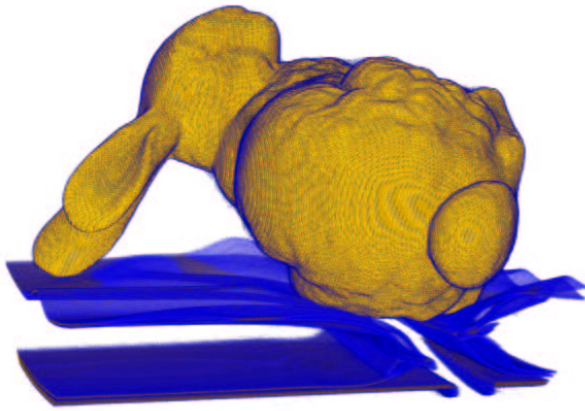
Therefore, future work includes the reduction of rendering artifacts, extensions of our approach to MIP mapping and deeper hierarchies, combinations with other shading techniques, and applications to more rendering techniques involving texture mapping.

Acknowledgements

Many people have contributed to this work; in particular, thanks to ATI Technologies for the Radeon 8500 graphics board, thanks to Klaus Engel for his volume rendering code, thanks to James Fowler for the QccPack software, thanks to the Stanford University Computer Graphics Laboratory for the CT scan data of the terra-cotta bunny available in the Stanford volume data archive and the buddha light field available in the Stanford light fields archive, thanks to Manfred Weiler, Daniel Weiskopf, Rüdiger Westermann, and again Klaus Engel for comments and discussions, and thanks to the anonymous reviewers for their comments and suggestions.

References

1. Chandrajit L. Bajaj, Insung Ihm, and Sanghun Park. Compression-Based 3D Texture Mapping for Real-Time Rendering. *Graphical Models*, **62**(6):391–410, 2000.
2. Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha. Rendering from Compressed Textures. In *Proceedings of SIGGRAPH '96*, pages 373–378, 1996.
3. Biran Cabral, Nancy Cam, and Jim Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proceedings of 1994 Symposium on Volume Visualization*, pages 91–97.
4. Randima Fernando, Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg. Adaptive Shadow Maps. In *Proceedings of SIGGRAPH 2001*, pages 387–390, 2001.
5. James E. Fowler. QccPack: An Open-Source Software Library for Quantization, Compression, and Coding. In *Applications of Digital Image Processing XXIII (Proceedings SPIE 4115)*, pages 294–301, 2000.
6. Allen Gersho and Robert M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, Boston, 1992.
7. Evan Hart and Jason L. Mitchell. *Hardware Shading with EXT_vertex_shader and ATI_fragment_shader*. ATI Technologies, 2002.
8. Paul S. Heckbert. Adaptive Radiosity Textures for Bidirectional Ray Tracing. *Computer Graphics (Proceedings of SIGGRAPH '90)*, **24**(4):145–154, 1990.
9. Mark J. Kilgard, Ed. *NVIDIA OpenGL Extension Specifications*. NVIDIA Corporation, 2001
10. Marc Levoy and Pat Hanrahan. Light Field Rendering. In *Proceedings of SIGGRAPH '96*, pages 31–42, 1996.
11. Paul Ning and Lambertus Hesselink. Vector Quantization for Volume Rendering. In *Proceedings of 1992 Workshop on Volume Visualization*, pages 69–74, 1992.
12. Paul Ning and Lambertus Hesselink. Fast Volume Rendering of Compressed Data. In *Proceedings of Visualization '93*, pages 11–18, 1993.
13. *S3TC DirectX 6.0 Standard Texture Compression*. S3 Incorporated, 1998.
14. Jay Torborg, James T. Kajiya. Talisman: Commodity Realtime 3D Graphics for the PC. In *Proceedings of SIGGRAPH '96*, pages 353–363, 1996.
15. Rüdiger Westermann, Leif Kobbelt, and Thomas Ertl. Real-time Exploration of Regular Volume Data by Adaptive Reconstruction of Isosurfaces. *The Visual Computer*, **15**(2):100–111, 1999.
16. Yizhou Yu, Andras Ferencz, and Jitendra Malik. Compressing Texture Maps for Large Real Environments. SIGGRAPH 2000 Sketch, 2000.



(a)



(b)

Figure 12: (a) Volume rendering of the $512 \times 512 \times 360$ Stanford terra-cotta bunny using vector quantization. (b) Light field rendering of a $16 \times 16 \times 256 \times 256$ variant of the Stanford buddha light field. Slight rendering artifacts are visible at cell boundaries (see Figure 11).