Compression-Based 3D Texture Mapping for Real-Time Rendering[1]

Chandrajit Bajaj† and Insung Ihm‡ and Sanghun Park§

†*Department of Computer Sciences, The University of Texas at Austin, U.S.A.; and* ‡*Department of Computer Science, Sogang University, Korea; and* §*TICAM, The University of Texas at Austin, U.S.A.*

Received ??; accepted ??

While 2D texture mapping is one of the most effective rendering techniques that make 3D objects appear visually interesting, it often suffers from visual artifacts produced when 2D image patterns are wrapped onto the surface of objects with arbitrary shapes. On the other hand, 3D texture mapping generates highly natural visual effects in which objects appear carved from lumps of materials rather than laminated with thin sheets as in 2D texture mapping. Storing 3D texture images in a table for fast mapping computations, instead of evaluating procedures on the fly, however, has been considered impractical due to the extremely high memory requirement. In this paper, we present a new effective method for 3D texture mapping designed for real-time rendering of polygonal models. Our scheme attempts to resolve the potential texture memory problem by compressing 3D textures using a wavelet-based encoding method. The experimental results on various non-trivial 3D textures and polygonal models show that high compression rates are achieved with few visual artifacts in the rendered images and a small impact on rendering time. The simplicity of our compression-based scheme will make it easy to implement practical 3D texture mapping in software/hardware rendering systems including the real-time 3D graphics APIs like OpenGL and Direct3D.

Key Words: texture mapping; 3D texture; data compression; wavelet; real-time rendering; OpenGL

1. INTRODUCTION

Texture mapping is one of the most powerful rendering techniques that make three-dimensional objects appear visually more complex and realistic [9]. Two-dimensional texture mapping has been popular in creating many interesting visual effects by projecting 2D image patterns onto the surface of solid objects. While it has proved very useful in adding realism in rendering, 2D texture mapping suffers from the limitation that it is

[1] This work has been supported in part by the Ministry of Information & Communication of Korea under University Foundation Research Program 2000.

often difficult to wrap 2D patterns, without visual artifacts, onto the surface of objects having complicated shapes. As an attempt to alleviate the computational complications of wrapping as well as to resolve the visual artifacts, Peachey [14] and Perlin [15] presented the use of space filling 3D texture images, called *solid textures*. Many of the textures found in nature such as wood, marble, and gases, are easily simulated with solid textures that map three-dimensional object space to color space [5]. Unlike 2D textures, they exist not only on the surface of objects but also inside the objects. Texture colors are assigned to any point of the entire solid object simply by evaluating the specified functions or codes according to their positions in 3D space. The 3D solid texture mapping can be viewed as immersing geometric objects in virtual volumes associated with 3D textures, and obtaining necessary texture colors from the solid textures. This 3D texture mapping produces highly natural visual effects in which objects appear carved from lumps of materials rather than laminated on the surfaces as in 2D texture mapping. The difference between 2D and 3D mappings is prominent particularly when objects have complicated geometry and topology since 3D textures are not visually affected by the distortions that exist in object parameter space.

Many useful 3D textures are generally synthesized procedurally instead of painting or digitizing them (Refer to [5] for several interesting examples.). They are based on mathematical functions or programs that take 3D coordinates of points as input, and compute their corresponding texture values. The evaluation is usually carried out on the fly during the rendering computation. While procedural texture models provide a very compact representation, evaluating procedural textures as necessary during texture mapping leads to slower rendering than accessing pre-sampled textures stored in simple arrays.

While using sampled 3D texture maps in 3D volumetric form is faster, they tend to take up a large amount of texture memory. For example, when a 3D RGB texture with resolution $256 \times 256 \times 256$ is represented in one byte per color channel, it requires 48 Mbytes (=50,331,648 bytes) of texture memory. Although some recent graphics systems allow the use of main memory for textures, such texture memory costs are an impossible burden on most current graphics systems. Storing several elaborate textures with higher resolution, say, $512 \times 512 \times 512$ would be prohibitive even to the most advanced rendering systems. Obviously, there is a tradeoff between the size of texture memory and the computation time. Explicitly storing sampled textures in dedicated memory, and fetching texture colors as necessary, as in the current graphics accelerator supporting real-time 2D texture mapping, can generate images faster than evaluating them on the fly. To make this feasible for 3D texture mapping, however, an efficient way of manipulating potentially huge textures needs to be invented.

This paper presents a new and practical scheme for real-time 3D texture mapping which is easily implemented. Our technique relies on 3D RGB volume compression and efficient processing of compressed solid textures. The idea of rendering directly from compressed textures has been presented first in [3], where they used vector quantization to compress 2D textures in simple or mipmap form. Texture compression saves memory space for storing textures as well as decreases the system bandwidth required for texturing, which allows more detailed textures to be used with improved performance. Recently, several 3D hardware accelerator vendors have adopted various compression techniques in implementing 2D texture mapping in hardware [1, 17, 21]. To compress 3D textures, we use a wavelet-based compression method that provides fast decoding to random data access, as well as fairly high compression rates [2]. This compression technique exploits the power of wavelet theory and naturally provides multi-resolution representations of 3D RGB volumes. With this compression method, we can store mipmaps for 3D textures of non-trivial resolutions very compactly in texture memory. Its fast random access decoding ability also results in only a small impact on rendering time. The simplicity of our new 3D texture mapping scheme makes it easy to implement in software/hardware rendering systems. Furthermore, 3D real-time graphics APIs like OpenGL and Direct3D can be extended with little effort to include 3D texture mapping without heavy demand for very large texture memory.

The rest of this paper is organized as follows: In Section 2, we provide a detailed description of the new compression-based 3D texture mapping technique. Experimental results on various 3D textures and polygonal objects are reported in Section 3, and the paper is concluded in Section 4.

2. A NEW 3D TEXTURE MAPPING SCHEME

In this section, we describe the new 3D texture mapping method suitable for real-time rendering of polygonal models. The idea presented here can also be used effectively in other rendering systems such as RenderMan [16] to enhance the texture mapping speed. The key point in our texture mapping scheme is to extract only the necessary portion from the full 3D texture map, then compress it in compact form where fast run-time decoding for random access to texels is possible. In particular, the compression method we apply is based on wavelet theory, and naturally supports multi-resolution representations of 3D textures. This capability of the compression method makes it easy to construct a 3D texture mipmap using a small amount of texture memory. Fig. 1 illustrates the 3D texture mapping pipeline in which the first three steps, *3D Texture Modeling*, *3D Texture Cell Selection*, and *3D Texture Compression* comprise the necessary pre-processing stages. In the following subsections, we provide detailed explanations of the various stages in the pipeline.

2.1. 3D Texture Modeling

Our scheme assumes, as an input texture, a sampled 3D RGB texture stored in a 3D array. It is generated by sampling texel values from a three-dimensional texture field that is usually described procedurally. The storage requirements are very high for uncompressed 3D texture images at reasonable resolution: 256^3 and 512^3 RGB textures need 48 Mbytes and 384 Mbytes, respectively. This is one of the reasons which make fast 3D texture mapping with stored textures appear impractical.

In the texture modeling stage, a polygonal object in its object space $R_{os} = \{(x, y, z) \mid -\infty < x, y, z < \infty\}$ is textured by putting it in a 3D texture defined in the texture space $R_{ts} = \{(s, t, r) \mid 0 \leq s, t, r \leq 1\}$, and finding the intersection of the object's surface and the solid texture. Texturing an object can be viewed as determining a function $f : R_{os} \longrightarrow R_{ts}$. This function f can be chosen arbitrarily.

2.2. 3D Texture Cell Selection

Once a mapping between a polygonal object and a 3D texture map is fixed, the unnecessary texture data is eliminated to reduce storage space. Consider an $n_s \times n_t \times n_r$ texture. In our scheme, the texture data is subdivided into small subblocks of size $n_c \times n_c \times n_c$, called *texture cells* (In the current implementation, the resolution of texture cell is $4 \times 4 \times 4$.). The texture cell is a basic unit for selecting texture data that is actually needed for rendering.

In this 3D texture cell selection stage, each polygon on the boundary of an object is 3D-scan-converted to find all the texture cells that intersect with the surface of the solid

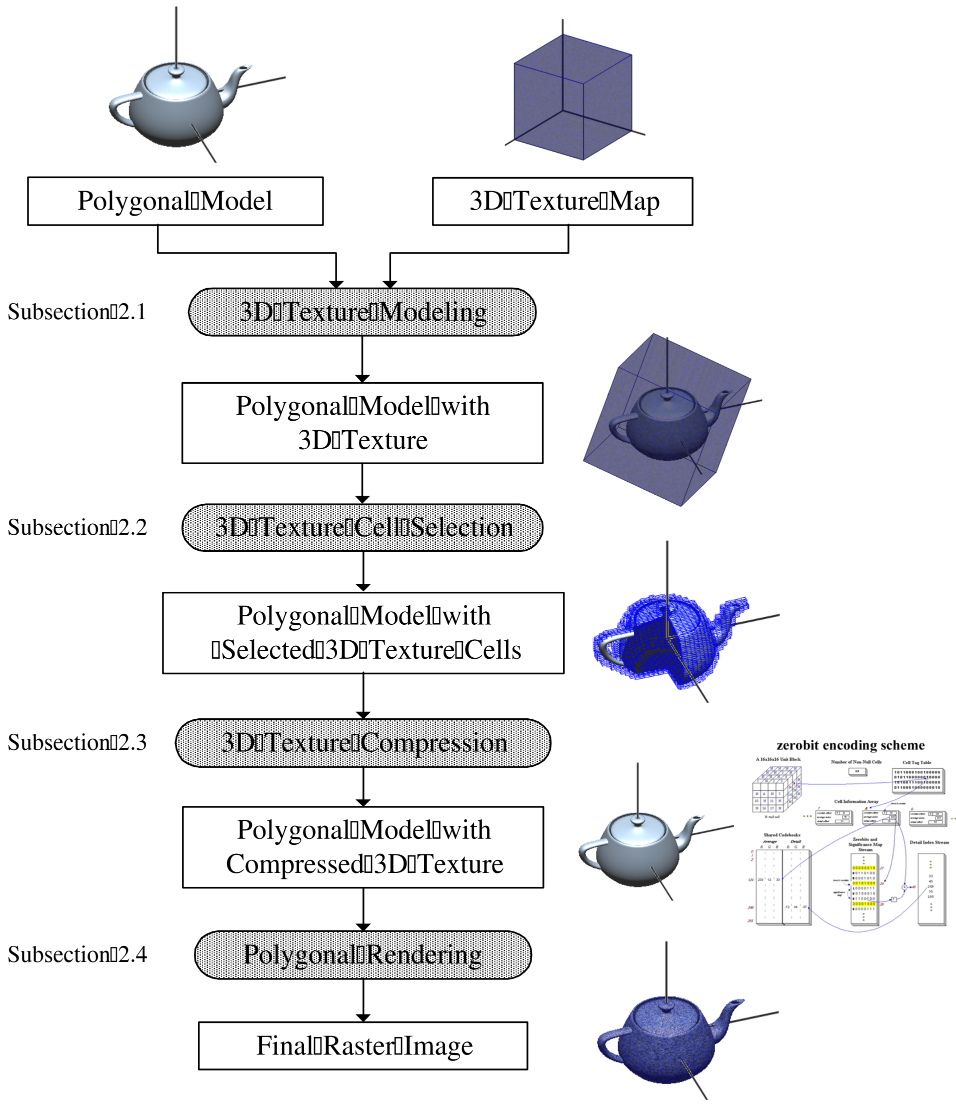


FIG. 1. Compression-based 3D texture mapping pipeline

object. Notice that texels in the selected texture cells contain all the texture information necessary for rendering. The cells that are not chosen are replaced by null cells, that is, cells with black color. By keeping nearby texels surrounding the surface of an object in this intermediate stage, a large portion of texture data is removed to alleviate the potential prohibitive storage requirement. The selected texture cells take only a small percentage of the original texture data. The null cells still exist in the texture map in this stage, and the texture size remains the same. However, the spatial coherence created by null cells makes an encoding scheme efficiently compress the 3D texture in compact form in the next stage.

2.3. 3D Texture Compression

2.3.1. Choosing an Appropriate Compression Technique

There exist many data compression methods for efficient storage and transmission. It is very important to choose a compression technique which is most appropriate for this specific 3D texture mapping application. We have several issues to consider as similarly discussed in [3, 11]:

1. **High compression rate and visual fidelity.** Non-trivial 3D textures are often very large in size, ranging from a few dozen megabytes to several hundred megabytes. When a mipmap is used for a pre-filtered multi-resolution representation, the size gets even larger. Developing real-time applications with such data assumes, implicitly or explicitly, that the entire data is loaded into main memory for efficient run-time processing. This places an enormous burden on storage space as well as transmission bandwidth. While lossless compression techniques preserve data without introducing reconstruction errors, they often fail to achieve compression rates high enough for practical implementation of 3D texture mapping. The loss of information associated with lossy compression methods, however, needs to be controlled properly as it is important to minimize the distortion in the reconstructed textures.

2. **Fast decoding for random access.** The general concern of most lossy compression schemes is achieving the best compression rate with minimal distortion in the reconstructed images [7, 18]. Such compression methods, however, often impose constraints on the random access decoding ability, which makes them inappropriate for real-time texture mapping applications where it is difficult to predict data access patterns in advance. For instance, variable-bitrate or differential encoding schemes such as Huffman or arithmetic coders coupled to block JPEG or MPEG schemes, do not lend themselves to efficiently decode individual texels that are accessed in a random pattern during run-time.

3. **Multi-resolution representation.** Mipmapping is the most commonly used anti-aliasing technique for 2D texture mapping [22]. A mipmap of a 2D texture is a pyramid of pre-filtered images obtained by averaging down the original image to successively lower resolutions. Mipmapping with level-of-detail representations of textures offers fast and constant filtering of texels, and its simplicity lends itself to an efficient hardware implementation. The idea naturally extends to 3D textures although mipmaps for 3D textures are considered even more impractical due to the additional memory requirement. It is highly recommended to choose a compression technique that provides a multi-resolution representation in its compression scheme.

4. **Exploitation of 3D data redundancy.** 3D textures are three-dimensional data that exhibits redundancy in all three dimensions. A compression scheme devised for 2D images could be applied to compress each slice in 3D textures, however, a good compression

technique must be able to fully exploit data coherence in all three dimensions to maximize the compression performance.

5. **Selective block-wise compression.** In some applications like ours, it is more efficient to selectively compress a certain portion of data rather than the entire dataset. It is very desirable that a compression scheme includes this selective compression capability in its encoding algorithm for the effective compression.

2.3.2. The Zerobit Encoding Scheme

The above five desirable characteristics are common to most real-time applications that must handle discrete sampled data of very large sizes. Vector quantization has been popular in developing such applications mainly because it supports fast random decoding through table lookups [6]. Some recent applications of vector quantization in the computer graphics field, include compression of CT/MRI datasets [12], light fields [11], and 2D textures [3]. Some 3D graphics accelerators, for example, the PowerVR architecture [21], adopted vector quantization for 2D texture mapping. Some other compression techniques have also been developed for compressing 2D texture maps. The S3 texture compression scheme S3TC, which became the basis for the compressed texture format used in DirectX 6.0, breaks a texture map in 4×4 blocks of texels [17]. Each block is stored with a 32 bit bitmap – 2 bits per texel, and two representative 16 bit colors. The two bit index of a texel points to a four color lookup table, made of the two explicitly encoded colors and two additional colors that are derived by uniformly interpolating the explicitly encoded colors. The FXT1 scheme of 3dfx also divides a texture image into 4×4 and/or 4×8 texel blocks [1]. It uses four different compression algorithms, one of which is similar to S3TC. In this scheme, the best algorithm is chosen per block to generate the highest quality result.

Recently, a new compression scheme for 3D RGB images has been developed as an alternative to vector quantization [2]. This technique, called *zerobit encoding*, is suitable for applications wherein data is accessed in an unpredictable manner, and real-time performance of decoding is required. It extends the idea of the compression scheme [10] for 3D gray-scale volume data to compression of 3D RGB images, and its new encoding structure significantly improves decompression speeds. Unlike vector quantization, the zerobit encoding scheme, based on the wavelet theory, naturally offers a multi-resolution representation for 3D images. Experimental results on test datasets show that this compression scheme provides fast random access to compressed data in addition to achieving fairly high compression rates.

Like other transform coding algorithms, the compression scheme consists of three major stages: transform, quantization and encoding. A 3D RGB image is first partitioned into $16 \times 16 \times 16$ blocks, called *unit blocks*. They are subdivided into $4 \times 4 \times 4$ blocks, called *cells*, to which the 3D Haar transform is applied twice to exploit data coherence in all of the three dimensions. The level of wavelet compression is controlled by specifying a target ratio λ of non-zero coefficients that survive the truncation. From this target ratio, the corresponding threshold value τ is computed where τ is the norm of the ($\lambda \cdot$ the total number of voxels)-th largest coefficient. After the transform, the wavelet coefficients with norm that is smaller than τ are truncated. Once the truncated coefficients are replaced by zeros, the non-zero wavelet coefficients are quantized into 8 bit indices with codebooks having 24 bit codewords. In the last stage of compression, the strings of symbols coming from the quantizer are losslessly encoded using the zerobit encoding technique, which supports fast decoding for random access to compressed 3D images (Fig. 2). As a result of two

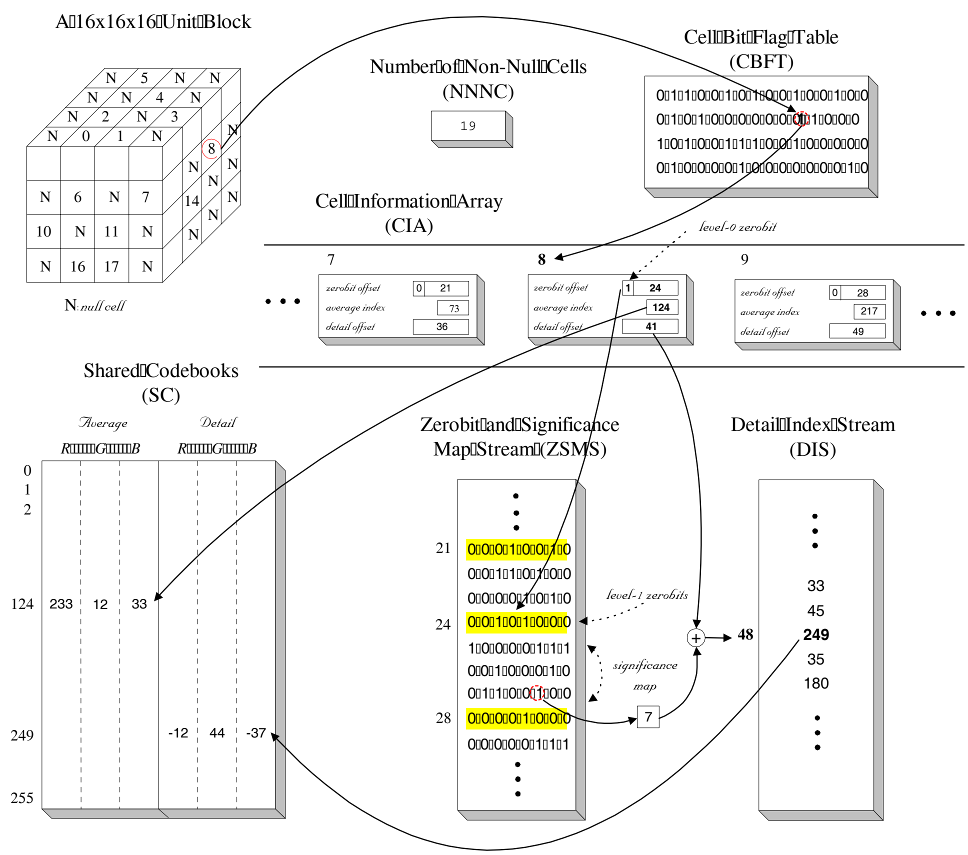


FIG. 2. The zerobit encoding scheme [2]

TABLE 1a

Comparisons of the Two Compression Schemes: Compression Rates and Fidelity [2]

		Vector Quantization	Zerobit Encoding 2%	3%	4%	5%
buddha	Size (MB)	8.81	2.11	2.90	3.63	4.31
	Comp. Rate	21.79	91.11	66.26	52.89	44.51
	PSNR (dB)	38.00	39.26	41.70	43.63	45.18
dragon	Size (MB)	9.52	2.31	3.15	4.09	5.02
	Comp. Rate	20.18	83.03	60.87	46.99	38.21
	PSNR (dB)	35.58	31.00	32.17	33.37	34.40

TABLE 1b

Comparisons of the Two Compression Schemes: Average Rendering Times (in Frames Per Second) [2]

		Vector Quantization	Zerobit Encoding 2%	3%	4%	5%
buddha	st-lerp	9.46	13.60	13.60	13.60	13.60
	uvst-lerp	2.68	2.99	2.98	2.98	2.98
dragon	st-lerp	17.55	24.60	24.44	24.20	23.97
	uvst-lerp	5.66	5.74	5.71	5.66	5.62

applications of the 3D Haar transform, one average coefficient, one set of seven detail coefficients on level 0, and 8 sets of seven detail coefficients on level 1 are generated that represent three levels of detail. In order to reconstruct a voxel value, the average, the details on level 0, and an appropriate set of details on level 1 are necessary. Since only 1 to 10 per cent of coefficients are usually used in compression, most detail coefficients are zeroed out after truncation, and the resulting null coefficients exist in thick clusters. The zerobits in the encoding scheme are flags that indicate whether each set of detail coefficients contains only null coefficients. When a set includes zero coefficients only, neither decoding of its seven details nor application of the inverse transform is necessary. The zerobit encoding scheme is designed to quickly determine null sets of detail nodes using zerobits, which provides large savings in the reconstruction computation. Refer to [2] for the details on the encoding scheme. Notice that the texture cell in our 3D texture mapping scheme naturally corresponds to the cell in this compression technique.

Table 1 shows sample statistics on the performance of the zerobit encoding and vector quantization used in [11] for two representative light field datasets buddha and dragon with resolution $32 \times 32 \times 256 \times 256$ (192Mbytes) [2]. To apply the zerobit encoding technique, the 4D sampled light field datasets were rearranged into 3D images, then were compressed. While the vector quantization yielded compression rates 21.79 and 20.18 for buddha and dragon, the zerobit encoding method produced higher rates of 44.51 to 91.11 and 38.21 to 83.03 at the selected four target ratios, respectively (Table 1a) [2]. The PSNR results show that the qualities of reconstructed images are about the same when about 2%

[2] These rates exclude the gzip compression, that could follow both compression methods for efficient storage as in [11].

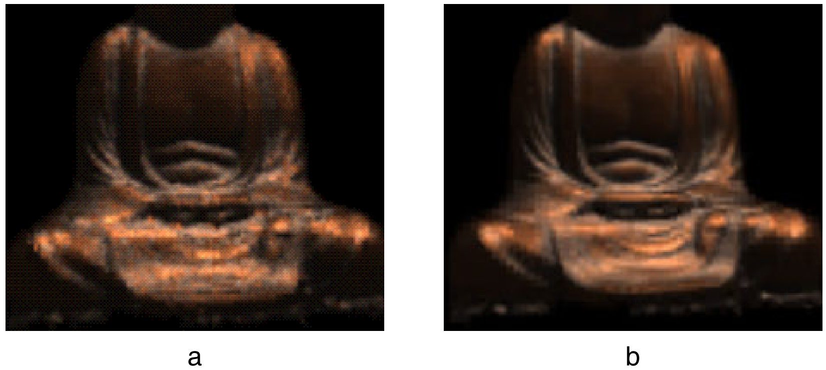


FIG. 3. Image-based rendered images: (a) vector quantization, (b) zerobit encoding (3% of wavelet coefficients used) [2]

and 5% of coefficients are used in zerobit encoding for the buddha and dragon datasets, respectively (See Fig. 3 for the portion of two sample buddha images.)[3].

The image-based rendering time, spent on displaying 76 frames of 382 × 382 pixels with gradually varying viewing parameters, was measured on an SGI workstation with a 195 MHz MIPS R10000 CPU. Two cases of bilinear interpolation on the st-plane (st-lerp) and quadralinear interpolation on both uv- and st-planes (uvst-lerp) were tested (Table 1b). The timing results show the zerobit encoding scheme generates more frames per second for both datasets in most cases. Note that the reconstruction cost per data item for vector quantization is very cheap since decompression is performed through a simple codebook lookup, and is cheaper than zerobit encoding on average. However, zerobit encoding decompresses several data items, 4 planes in this case, at the same time, and is very quick particularly when data in empty background regions is reconstructed, which results in the overall faster rendering.

While the empirical comparisons for a few applications can not prove that the zerobit encoding method is always superior to vector quantization, we find the former compares very favorably to the latter. In our 3D texture mapping technique, we use the zerobit encoding scheme to compress the selected texture cells. As will be explained in the next section, it also turns out to be very effective in compressing 3D textures.

2.4. Polygonal Rendering with Compressed Textures

2.4.1. A New Capability for OpenGL 1.2

When applying textures to geometric objects, the necessary texel values are repeatedly fetched from zerobit-encoded 3D textures using their texture coordinates. The compression-based 3D texture mapping can enhance the rendering speed in any rendering method including time-consuming photo-realistic rendering. In our implementation, we applied our scheme to real-time rendering and extended the OpenGL library to include the feature of 3D texture mapping with zerobit-encoded textures. Note that 3D texture mapping has been a commonly available extension to several vendor's OpenGL 1.1 implementations, and is now

---

[3]The mean-square peak-signal-to-noise ratio (PSNR) is defined as PSNR (dB) = $10 \log_{10} \frac{x_{peak}^2}{\sigma^2}$ where x_{peak}^2 is the peak value of the signal, and σ^2 is the mean squared error. It is one of the frequently used objective fidelity measures that indicates the size of the error relative to the peak value of the signal.

one of the core capabilities that must be supported by all OpenGL 1.2 implementations [19]. The **glTexImage1D()** and **glTexImage2D()** functions are extended for 3D texture mapping where the command for specifying a three-dimensional texture image is defined as

`void` **glTexImage3D** (`GLenum` *target*, `GLint` *level*, `GLint` *internalformat*, `GLsizei` *width*, `GLsizei` *height*, `GLsizei` *depth*, `GLint` *border*, `GLenum` *format*, `GLenum` *type*, `const GLvoid` **texels*);

With *target* `GL_TEXTURE_3D`, this command reads a texture of size $width \times height \times depth$, that is stored in memory, pointed by *texels*, in *internalformat*. For a compressed texture, our extension adds a symbolic constant `GL_UNSIGNED_BYTE_COMPRESSED` for the parameter *type* to read a compressed texture, whose texels are stored in unsigned character, on levels *level*, *level*+1, and *level*+2.

When 3D texture mapping is enabled by calling **glEnable**(`GL_TEXTURE_3D`), and a compressed 3D texture is specified, the texture is assumed to be in compressed form, and texels are fetched from the zerobit-encoded structure rather than a simple array. The extension is easy to implement since the new capability can be included simply by adding proper state variables and decoding functions. Other utility functions, such as creating encoded 3D textures with user-specified compression rates, could also be included in the OpenGL Utility Library (GLU).

2.4.2. Compact Representation of 3D Mipmaps

A 3D mipmap is an ordered set of 3D arrays representing the same texture where each successive array has a resolution lower than its previous one. 3D mipmapping is easily included into our scheme since mipmaps as well as single 3D textures are represented very compactly. Given a base 3D texture, the zerobit-encoded structure represents three levels of detail with level number 0, 1, and 2. The reduced images on the next three levels can be stored in another zerobit-encoded structure. An alternative is to store the texture images with lower resolutions except on level 0, 1, and 2, in simple 3D arrays. The images on the higher levels take up only a small amount of storage. For example, when a $256 \times 256 \times 256$ RGB texture image in unsigned character is loaded, the entire reduced images on levels $3, 4, \cdots, 8$ require only about 110 ($\approx 3\{(2^5)^3 + (2^4)^3 + \cdots (2^0)^3\}$) Kbytes in total.

2.5. Sharing of a 3D Texture between Multiple Objects

When a texture is compressed object by object, it could lead to a waste of texture memory. That is, if a 3D texture is shared by multiple polygonal objects, the same 3D texture cells can be replicated for several objects. We have been extending our method to support three types of compression modes: The first mode, called zerobit_encoding_single_object is one we have described in this paper. The second mode zerobit_encoding_multiple_objects is for the case in which several polygonal objects share a common 3D texture image. In this mode, all the 3D texture cells that are used by at least one object are selected before encoding. The last mode zerobit_encoding_entire_texture handles the dynamic situation in which it is difficult or impossible to predict which texture cells shall be used for rendering. For instance, an interesting animation can be generated by making an object float in a texture field, dynamically binding texture coordinates. In this case, the first two compression modes are not appropriate. The third mode compresses the entire 3D texture and loads it for rendering. While it is the most expensive one, this mode provides flexibility in texture mapping.

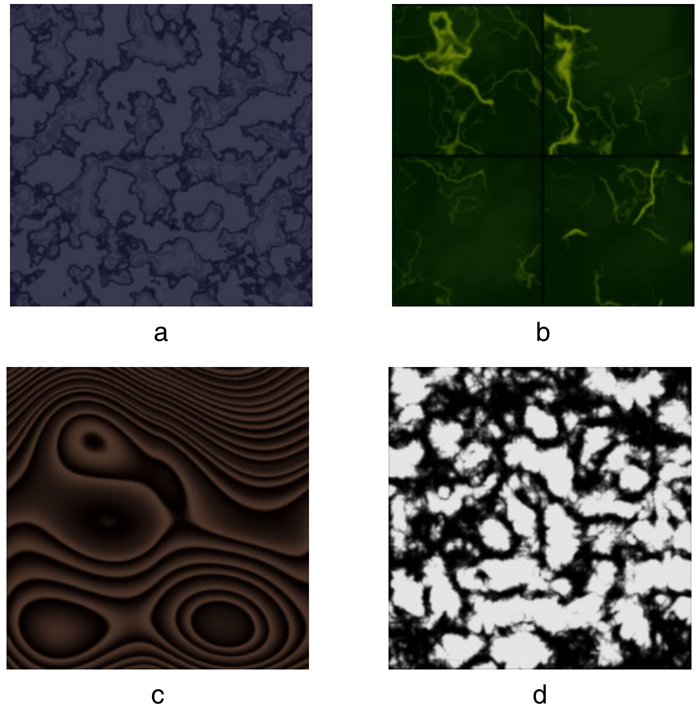


FIG. 4. Sample slices from the four example 3D textures: (a) Bmarble, (b) Gmarbpol, (c) Wood, (d) Eroded

3. EXPERIMENTAL RESULTS

3.1. Test Datasets

We have implemented our new 3D texture mapping scheme by extending the MESA 3D Graphics Library which is a publicly available OpenGL implementation [13]. The current version 3.0 supports the 3D texture mapping feature where the entire texture image is stored in a simple array without any compression. We added the necessary state variables and functions to handle zerobit-encoded 3D texture maps.

We have generated four different 3D texture images of size $256 \times 256 \times 256$ (Fig. 4). The texture images have three channel RGB colors, and their sizes amount to 48 Mbytes, respectively. The three textures Bmarble, Wood, and Eroded were created using the RenderMan surface shaders `blue_marble()`, `wood()`, and `eroded()`, respectively [20]. The surface shader `gmarbtile_polish()` for the texture Gmarbpol was written by Larry Gritz, and is available as a part of the Blue Moon Rendering Tools (BMRT). Our 3D texture mapping technique has been applied to several polygonal models with various shapes and sizes, including those listed in Table 2. The teapot model Teapot was polygonized from a parametric equation. The model Dragon and the next three models Bunny, Sdragon and Buddha were obtained from Viewpoint and the Stanford 3D Scanning Repository, respectively. Lastly, the model Head was created by generating an iso-surface from the UNC CT scan of a human head. The table shows how many $4 \times 4 \times 4$ texture cells are selected from the entire 262,144 ($= 64 \times 64 \times 64$) cells in $256 \times 256 \times 256$ textures through the 3D texture cell selection stage. In general, the ratios of selected cells are quite small. The rate is a little high for Head since the polygonal model has a complicated internal structure as a result of iso-surfacing.

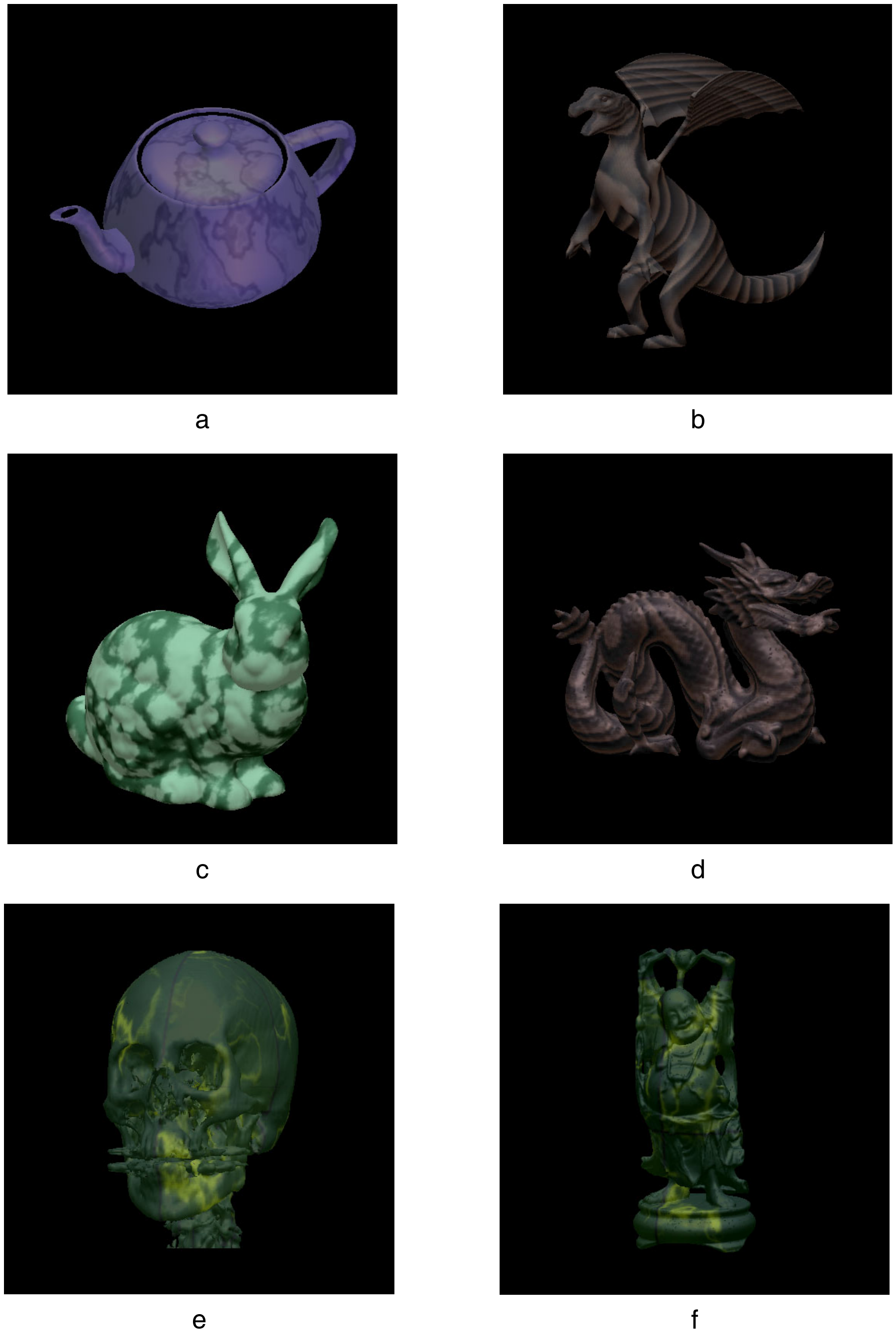


FIG. 5. Images rendered with `GL_LINEAR` from compressed textures (10%): (a) Teapot with Bmarble, (b) Dragon with Wood, (c) Bunny with Eroded, (d) Sdragon with Wood, (e) Head with Gmarbpol, (f) Buddha with Gmarbpol

TABLE 2

Ratios of Selected Texture Cells

Object	# of Faces	# of Selected Cells	Ratio (%)
Teapot	1,152	7,836	3.0
Dragon	12,078	7,965	3.0
Bunny	69,451	16,137	6.2
Sdragon	202,520	11,950	4.6
Head	203,544	30,881	11.8
Buddha	293,232	9,600	3.7

TABLE 3a

Sizes of Compressed Textures: Bmarble and Gmarbpol (256^3)

		Bmarble		Gmarbpol	
Object	Target Ratio	Size (KB)	Comp. Rate	Size (KB)	Comp. Rate
Entire	3%	1154	42.6	1166	42.2
	5%	1666	29.5	1602	30.7
	10%	2814	17.5	2502	19.7
Teapot	3%	190	258.7	190	258.7
	5%	226	217.5	210	234.1
	10%	290	169.5	246	199.8
Dragon	3%	182	270.1	174	282.5
	5%	222	221.4	198	248.2
	10%	278	176.8	238	206.5
Bunny	3%	258	190.5	238	206.5
	5%	326	150.8	278	176.8
	10%	466	105.5	346	142.1
Sdragon	3%	220	223.4	210	234.1
	5%	276	178.1	238	206.5
	10%	360	136.5	286	171.9
Head	3%	318	154.6	310	158.6
	5%	422	116.5	378	130.0
	10%	626	78.5	518	94.9
Buddha	3%	202	243.3	194	253.4
	5%	234	210.1	218	225.5
	10%	306	160.6	274	179.4

3.2. Performances

To find out how compactly these 3D textures can be associated with the polygonal objects, we compressed selected texture cells for the entire 28 combinations as shown in Table 3. In the zerobit encoding scheme, a user specifies a ratio of wavelet coefficients to be used after truncation in order to control the degree of compression [2]. The number, shown in the “Target Ratio" field of the tables, represents an approximate ratio of wavelet coefficients that are actually used in encoding. We compressed 3D textures at three target ratios 3%, 5%, and 10%, and rendered the polygonal objects with these compressed textures. In these tables, we compare sizes and compression rates for various cases where “Entire" is for the zerobit_encoding_entire_texture mode, and the others for the zerobit_encoding_single_object mode. Observe that it took less than 1 Mbytes of memory across all combinations, ranging from 174 Kbytes to 686 Kbytes when the single object mode was used. Considering that the size of the original textures is 48 Mbytes, we see that very high compression rates are indeed achieved through texture cell selection and zerobit encoding.

TABLE 3b

Sizes of Compressed Textures: Wood and Eroded ($\mathbf{256^3}$)

Object	Target Ratio	Wood Size (KB)	Wood Comp. Rate	Eroded Size (KB)	Eroded Comp. Rate
Entire	3%	1282	38.3	1218	40.4
	5%	1818	27.0	1726	28.5
	10%	3006	16.4	2882	17.1
Teapot	3%	194	253.4	194	253.4
	5%	230	213.7	226	217.5
	10%	318	154.6	298	165.0
Dragon	3%	190	258.7	190	258.7
	5%	230	213.7	230	213.7
	10%	310	158.6	298	165.0
Bunny	3%	274	179.4	270	182.0
	5%	342	143.7	334	147.2
	10%	510	96.4	486	101.1
Sdragon	3%	222	221.4	230	213.7
	5%	278	176.8	278	176.8
	10%	390	126.0	390	126.0
Head	3%	330	149.0	330	149.0
	5%	438	112.2	438	112.2
	10%	686	71.7	670	73.4
Buddha	3%	206	238.6	206	238.6
	5%	246	199.8	246	199.8
	10%	334	147.2	330	149.0

Fig. 5 shows sample images rendered with the linear filter `GL_LINEAR` from the compressed textures having a target ratio 10%. When the 3D textures are compressed with target ratios higher than 10%, the texture-mapped images, produced with the linear filter, are almost free of aliasing artifacts which are often caused by the loss of information during lossy compression. In Fig. 6, we enlarged a portion of the Bunny images to make the compression artifacts more visible. When the ratio is 3%, the blocky artifacts are clearly visible, but most features are still preserved well enough for many real-time applications such as 3D games and animation.

In order to check the timing performances, we measured the running time, spent on rendering 54 frames of 512×512 pixels with incrementally varying viewing parameters. They include all computations for rendering including 3D texture mapping, view parameter setting, and displaying the final images. The timings were measured on an SGI Octane workstation with a 195 MHz R10000 CPU and 256 Mbytes of memory without hardware graphics acceleration. Table 4 reports the average time per frame in seconds for three difference rendering modes in which 48 Kbytes of texture cache was used (See the discussion on texture caching in Subsection 3.4.). The “GSO" field in this table is the time taken for rendering the objects using Gouraud shading only, and indicates how complicated is the involved rendering. Then, our new compression-based texturing scheme was compared with texture mapping without compression to evaluate overheads for fetching texels from compressed textures. Two filtering methods `GL_NEAREST` and `GL_LINEAR` were tested whose performances are shown in the “3DTMN" and “3DTML" fields, respectively. The

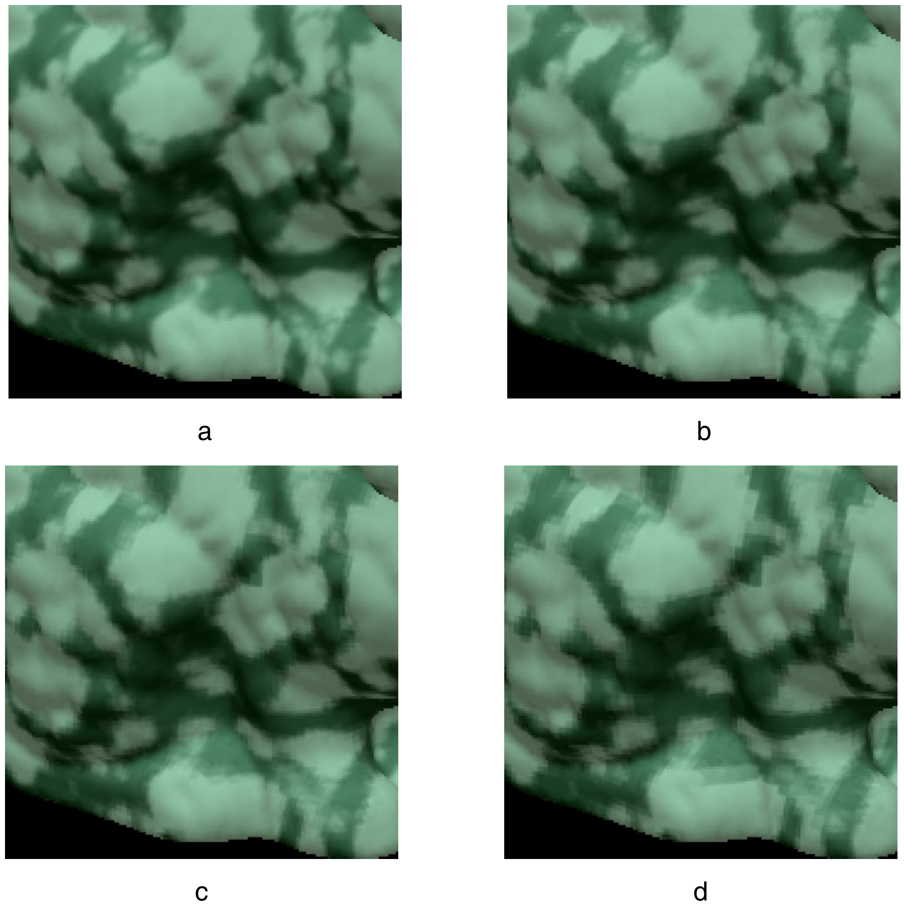


FIG. 6. Aliasing artifacts of compression-based 3D texture mapping (2X): (a) uncompressed (48 MB), (b) compressed (10%, 486 KB), (c) compressed (5%, 334 KB), (d) compressed (3%, 270 KB)

running time is generally proportional to the number of pixels that objects are projected into. As indicated by the test results, the zerobit encoding method provides very fast decoding speeds. We observe only a 8 percent and a 9 percent impact on rendering time on average for the nearest and the linear filter, respectively. Notice that the linear filtering method takes, for instance, 0.43 second to render Teapot from its uncompressed texture of size 48 Mbytes. On the other hand, the same filtering takes 0.51 second to produce a Teapot image with few visual artifacts from its compressed texture of size 290 Kbytes (target ratio = 10%). The benefit from our compression-based 3D texture mapping is evident, and is critical in particular when the texture memory resource is rather limited.

We have also generated two more elaborate textures of $512 \times 512 \times 512$ whose sizes are 384 Mbytes, and tested our texture mapping scheme with these huge textures (Table 5). The experiments indicate that 510 Kbytes to 1.70 Mbytes of memory are required to store the textures compressed at the target ratios 3%, 5%, and 10%, achieving compression rates of 225.7 to 771.0 . Compared to the 256^3 textures, compression-based renderings take 1.32 (Teapot with Bmarble) and 1.14 (Head with Gmarbpol) times as long on the average for the 512^3 textures. We were not able to load the entire uncompressed textures for rendering onto our workstation with 256 Mbytes of main memory, but expect that the rendering times will also get slower at the same rate.

Fig. 7 makes a comparison between renderings with four different texture mapping parameters. When Teapot is rendered from the 512^3 texture with a target ratio of 10% and the linear filter (Fig. 7b), the texture pattern on the surface appears much clearer than in the image, produced from the uncompressed 256^3 texture with the same filter (Fig. 7a). When the faster but inferior nearest filter is applied to the 512^3 texture with a target ratio 5% or 10% (Fig. 7d), consuming 0.23 second and 618 Kbytes (5%), or 0.26 second and 810 Kbytes (10%), respectively, the qualities are superior to the case in which the slower but better linear filter is applied to the 256^3 texture with a target ratio 10%, requiring 0.51 second and 290 Kbytes. Obviously, there is a tradeoff between rendering time, image quality, and memory requirement, and a choice of various texture mapping parameters should be made to optimize the application's needs.

3.3. Implementation of 3D Mipmapping

Implementing the mipmapping minimization filter involves two important tasks: One is how to represent the mipmap of a 3D texture internally, and the other is how to determine the level-of-detail factor d that indicates the level of reduced image to be applied. As explained in Subsection 2.4.2, the zerobit encoding scheme represents three levels of detail in its encoded structure, hence provides an effective way of 3D mipmap representation. Computing d can be done by naturally extending the measure used in the 2D mipmapping. Fig. 8a shows an example rendering of zerobit-encoded Bunny with levels of detail 0, 1, and 2, where the detail measure d is colored using a linearly varying color map in Fig. 8b.

TABLE 4

Average Rendering Times (in Seconds): GSO - Gouraud Shading Only, 3DTMN - 3D Texture Mapping (Nearest), 3DTML - 3D Texture Mapping (Linear)

Object & Texture	Target Ratio	GSO	3DTMN	3DTML
Teapot	uncomp.	0.05	0.15	0.43
with Bmarble	3%	–	0.16	0.48
	5%	–	0.17	0.49
	10%	–	0.18	0.51
Dragon	uncomp.	0.27	0.50	0.97
with Wood	3%	–	0.53	1.06
	5%	–	0.55	1.09
	10%	–	0.58	1.13
Bunny	uncomp.	1.03	1.44	1.93
with Eroded	3%	–	1.61	2.12
	5%	–	1.65	2.18
	10%	–	1.72	2.32
Sdragon	uncomp.	3.11	3.86	4.13
with Wood	3%	–	3.87	4.31
	5%	–	3.89	4.37
	10%	–	3.92	4.44
Head	uncomp.	2.98	3.90	4.70
with Gmarbpol	3%	–	4.10	4.79
	5%	–	4.14	4.85
	10%	–	4.22	4.96
Buddha	uncomp.	4.40	5.04	5.38
with Gmarbpol	3%	–	5.10	5.57
	5%	–	5.12	5.59
	10%	–	5.12	5.64

TABLE 5a

Experimental Results on 512^3 Textures: Sizes of Compressed Textures

Object & Texture	Target Ratio	Size (KB)	Comp. Rate
Teapot with Bmarble	3%	510	771.0
	5%	618	636.3
	10%	810	485.5
Head with Gmarbpol	3%	1110	354.3
	5%	1358	289.6
	10%	1742	225.7

TABLE 5b

Experimental Results on 512^3 Textures: Average Rendering Times (in Seconds): GSO - Gouraud Shading Only, 3DTMN - 3D Texture Mapping (Nearest), 3DTML - 3D Texture Mapping (Linear)

Object & Texture	Target Ratio	GSO	3DTMN	3DTML
Teapot with Bluemarble	uncomp.	0.05	–	–
	3%	–	0.21	0.60
	5%	–	0.23	0.62
	10%	–	0.26	0.66
Head with Gmarbpol	uncomp.	2.98	–	–
	3%	–	4.39	5.61
	5%	–	4.50	5.77
	10%	–	4.66	6.00

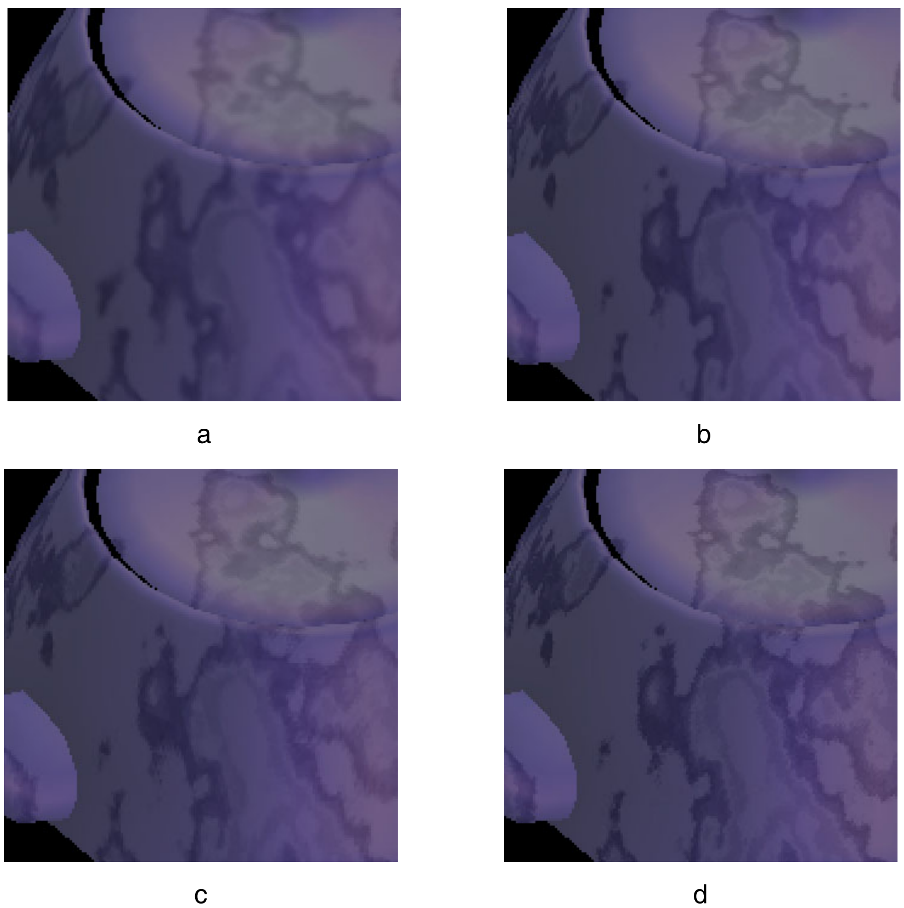
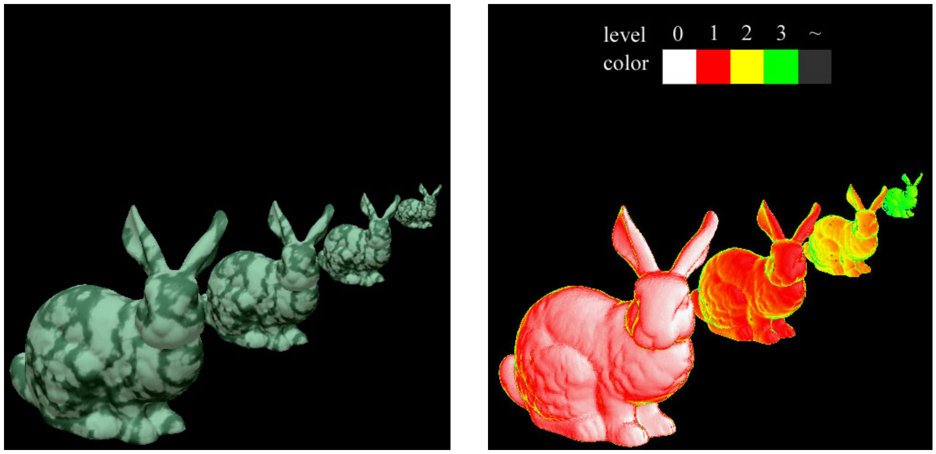


FIG. 7. Comparison between renderings with 256^3 and 512^3 textures (2X): (a) 256^3 (uncompressed & linear, 48 MB), (b) 512^3 (10% & linear, 810 KB), (c) 512^3 (3% & linear, 510 KB), (d) 512^3 (10% & nearest, 810 KB)



a b

FIG. 8. 3D mipmapping with zerobit encoding: (a) mipmapped Bunny, (b) mipmap levels of detail

3.4. Texture Caching

Although the zerobit encoding scheme offers fast reconstruction of texel values, texture caching can improve the rendering performance by exploiting the locality property of texel reference [8, 4]. In our scheme, when a texel value is necessary, all texels in the $4 \times 4 \times 4$ texture cell containing it is simultaneously reconstructed for efficiency. Rather than instantly throwing away used decompressed cells, storing them in a cache for the later use can possibly saves decoding computations. In order to see how texture caching affects the rendering performance, we experimented with a simple caching scheme. The texture cache we used is a circular list of cells where they are pre-empted with an LRU replacement policy. Note that each cell takes up 192 ($= 4 \times 4 \times 4 \times 3$) bytes.

Table 6 presents the timings when the 3D linear filter was used over the various cache sizes: 0 KB (no cache), 12 KB (64 cells), 24 KB (128 cells), 48 KB (256 cells), 96 KB (512 cells), and 192 KB (1024 cells). We tested with four representative combinations of polygonal objects and 3D textures using the same rendering parameters as in Subsection 3.2. When a fragment is textured with the linear filter, eight adjacent texels must be accessed. Thus, there exists a significant amount of spatial locality as adjacent fragments generated from polygons are rendered. Furthermore, there is an additional temporal locality of texel reference since 54 incrementally varying frames are generated in the test. It is shown that the hit rates are quite high for all tested cases, implying that the actual amount of texture cells actively in use at a particular time, is relatively small compared with the total compressed texture cells. We observe that the hit rates are particularly high across all the tested cache sizes when objects have a modest number of polygons like Teapot and Dragon (See Table 2 again.) In such a case, the effect of caching is prominent since texels are simply fetched from the cache most of the time rather than decompressed from

TABLE 6

The Effects of Cache Size: Average Hit Rates & Rendering Times (in Seconds)

Object & Texture	Target Ratio	No Cache	Cache Size 12KB	24KB	48KB	96KB	192KB
Teapot with Bmarble	Hit Rate (%)	–	98.93	99.03	99.10	99.11	99.13
	3%	1.29	0.48	0.48	0.48	0.49	0.49
	5%	1.62	0.49	0.49	0.49	0.50	0.50
	10%	2.23	0.51	0.51	0.51	0.51	0.51
Dragon with Wood	Hit Rate (%)	–	98.33	98.45	98.59	98.67	98.73
	3%	2.35	1.07	1.07	1.06	1.06	1.05
	5%	2.97	1.10	1.10	1.09	1.08	1.08
	10%	4.09	1.15	1.15	1.13	1.13	1.12
Bunny with Eroded	Hit Rate (%)	–	92.35	93.12	94.54	95.65	96.72
	3%	2.98	2.17	2.16	2.12	2.06	2.03
	5%	3.41	2.30	2.24	2.18	2.11	2.07
	10%	4.28	2.46	2.41	2.32	2.21	2.15
Buddha with Gmarbpol	Hit Rate (%)	–	92.19	93.57	96.76	98.47	98.71
	3%	6.31	5.57	5.61	5.57	5.48	5.40
	5%	6.63	5.65	5.67	5.59	5.49	5.41
	10%	7.18	5.75	5.77	5.64	5.52	5.44

zerobit-encoded textures. As the number of polygons increases as in Bunny and Buddha, the hit rates decrease in which case a larger cache size usually results in faster rendering. From the test result, we conclude that relatively small texture caches, say, 12 KB to 48 KB, are effective enough in our 3D texture mapping scheme.

4. CONCLUDING REMARKS

In this paper, we have presented a very effective method for 3D texture mapping, designed for real-time rendering of polygonal models. Our scheme attempts to resolve the potential texture memory problem arising from the very large sizes of 3D images by compressing them using the zerobit encoding scheme. This compression scheme not only provides fairly high compression rates but also offers very fast random access to individual texels. The experimental results on various non-trivial 3D textures and polygonal objects show that high compression rates are achieved with a small impact on rendering time and few visual artifacts in the rendered images. The simplicity of our compression-based 3D texture mapping scheme will make it easy to implement in software/hardware rendering systems. Currently, we are coding the auxiliary routines that are necessary for easy pre-processing. Once this is done, 3D real-time graphics APIs like OpenGL and Direct3D will be extended with little effort to include 3D texture mapping without heavy demand for texture memory.

ACKNOWLEDGEMENTS

We would like to thank Kiju Park and Joongyeon Lee for their help with experiments. The MESA 3D Graphics Library is an OpenGL implementation written by Brian Paul. We wish to thank the Stanford Graphics Lab., the UNC Graphics Lab., Viewpoint, and Larry Gritz for their data and codes.

REFERENCES

1. 3dfx Interactive. FXT1$^{\mathrm{TM}}$ texture compression technology. White Paper, 1999.
2. C. Bajaj, I. Ihm, and S. Park. 3D RGB image compression for interactive applications. Technical Report 99-41, TICAM, The Univ. of Texas at Austin, October 1999.
3. A. Beers, M. Agrawala, and N. Chaddha. Rendering from compressed texture. *Computer Graphics (Proc. SIGGRAPH '96)*, pages 373–378, 1996.
4. M. Cox, N. Bhandari, and M. Shantz. Multi-level texture caching for 3D graphics hardware. In *Proceedings of the 25th Annual International Symposium on Computer Architecure*, pages 86–97, June 1998.
5. D.S. Ebert, F.K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling: A Procedural Approach*. AP Professional, second edition, 1998.
6. A. Gersho and R.M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, 1992.
7. R. Gonzalez and R. Woods. *Digital Image Processing*. Addison-Wesley, 1993.
8. Z. Hakura and A. Gupta. The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th Annual International Symposium on Computer Architecure*, pages 108–120, June 1997.
9. P.S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, 1986.
10. I. Ihm and S. Park. Wavelet-based 3D compression scheme for interactive visualization of very large volume data. *Computer Graphics Forum*, 18(1):3–15, 1999.
11. M. Levoy and P. Hanrahan. Light field rendering. *Computer Graphics (Proc. SIGGRAPH '96)*, pages 31–42, 1996.
12. P. Ning and L. Hesselink. Fast volume rendering of compressed data. In *Proceedings of Visualization '93*, pages 11–18, San Jose, October 1993.
13. B. Paul. *The Mesa 3D Graphics Library*. $http://www.mesa3d.org$, 1999.
14. D.R. Peachey. Solid texturing of complex surfaces. *Computer Graphics (Proc. SIGGRAPH '85)*, 19(3):279–286, 1985.
15. K. Perlin. An image synthesizer. *Computer Graphics (Proc. SIGGRAPH '85)*, 19(3):287–296, 1985.
16. Pixar. *The RenderMan® Interface (Version 3.1)*, September 1989.
17. S3 Savage3D. S3TC$^{\mathrm{TM}}$ DirectX 6.0 standard texture compression. White Paper, 1999.
18. K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers, Inc., 1996.
19. M. Segal and K. Akeley. *The OpenGL® Graphcis System: A Specification (Version 1.2)*. Silicon Graphics, Inc., March 1998.
20. S. Upstill. *The RenderMan$^{\mathrm{TM}}$ Companion*. Addison-Wesley, 1990.
21. VideoLogic/NEC. PVRSG: PowerVR Second Generation. http://www.pvr-net.com, 1998.
22. L. Williams. Pyramidal parametrics. *Computer Graphics (Proc. SIGGRAPH '83)*, 17(3):1–11, 1983.