Rendering from Compressed Textures

*Andrew C. Beers, *Maneesh Agrawala, and †Navin Chaddha

* Computer Science Department
Stanford University

†Computer Systems Laboratory
Stanford University

Abstract

We present a simple method for rendering directly from compressed textures in hardware and software rendering systems. Textures are compressed using a vector quantization (VQ) method. The advantage of VQ over other compression techniques is that textures can be decompressed quickly during rendering. The drawback of using lossy compression schemes such as VQ for textures is that such methods introduce errors into the textures. We discuss techniques for controlling these losses. We also describe an extension to the basic VQ technique for compressing mipmaps. We have observed compression rates of up to $35:1$, with minimal loss in visual quality and a small impact on rendering time. The simplicity of our technique lends itself to an efficient hardware implementation.

CR categories: I.3.7 [Computer Graphics]: 3D Graphics and Realism - Texture; I.4.2 [Image Processing]: Compression - Coding

1 Introduction

Texture mapping is employed on high-end graphics workstations and rendering systems to increase the visual complexity of a scene without increasing its geometric complexity[7]. Texture mapping allows a rendering system to map an image onto simple scene geometry to make objects look much more complex or realistic than the underlying geometry. Recently, texture mapping hardware has become available on lower-end workstations, personal computers, and home game systems.

One of the costs of texture mapping is that the texture images often require a large amount of memory. For a particular scene, the memory required by the textures is dependent on the number of textures and the size of each texture. In some cases, the size of the textures may exceed the size of the scene geometry[3].

In hardware systems supporting real–time texture mapping, textures are generally placed in dedicated memory that can be accessed quickly as pixels are generated. In some hardware systems, textures are replicated in memory to facilitate fast parallel access [1]. Because texture memory is a limited resource in these systems, it can be consumed quickly. Although memory concerns are less severe for software rendering systems since textures are stored in main memory, there are advantages to conserving texture memory. In particular, using less memory for textures may yield caching benefits, especially in cases where the textures do not fit in main memory and cause the machine to swap.

*Gates Hall, Stanford University, Stanford, CA 94305
maneesh@cs.stanford.edu
http://www-graphics.stanford.edu/

One way to alleviate these memory limitations is to store compressed representations of textures in memory. A modified renderer could then render directly from this compressed representation. In this paper we examine the issues involved in rendering from compressed texture maps and propose a scheme for compressing textures using vector quantization (VQ)[4]. We also describe an extension of our basic VQ technique for compressing mipmaps. We show that by using VQ compression, we can achieve compression rates of up to $35:1$ with little loss in the visual quality of the rendered scene. We observe a 2 to 20 percent impact on rendering time using a software renderer that renders from the compressed format. However, the technique is so simple that incorporating it into hardware should have very little impact on rendering performance.

2 Choosing a Compression Scheme

There are many compression techniques for images, most of which are geared towards compression for storage or transmission. In choosing a compression scheme for texture mapping there are several issues to consider. In this section we discuss these issues and we show how VQ compression addresses each of them.

Decoding Speed. In order to render directly from the compressed representation, an essential feature of the compression scheme is fast decompression so that the time necessary to access a single texture pixel is not severely impacted. With VQ, decompression is performed through table lookups and is very fast. A transform coding scheme such as JPEG[10] is more expensive because extracting the value of a texture pixel would require an expensive inverse Discrete Cosine Transform (DCT) computation.

Random Access. It is difficult to know in advance how a renderer will access a texture. Thus, texture compression schemes must provide fast random access to pixels in the texture. For compression schemes like JPEG or run length coding which produce variable rate codes, extracting a texture pixel might require decompressing a large portion of the texture. Unlike variable rate codes, fixed-rate VQ represents each block of texture pixels with a fixed number of bits. Since the number of bits is known in advance, indexing any particular pixel is fast and easy.

Compression Rate and Visual Quality. While lossless compression schemes, such as Lempel–Ziv compression[12], will perfectly preserve a texture, they achieve much lower compression rates than lossy schemes. However, using a lossy compression scheme introduces errors into the textures. With VQ, there are many parameters that can be used to control these errors. A major difference between images and textures is that images are viewed on their own, while textures are viewed as part of a scene with orientation and size dependent on the mapping from scene surface to texture. Thus, for image compression algorithms, the visual quality of the compressed image is most important, while for texture compression algorithms, the visual quality of the rendered scene, not the texture map, is most important.

Encoding Speed. Experimenting with the compression rate versus visual quality tradeoff can be difficult if encoding is slow. Although optimal VQ encoding can be a time-consuming process, fast sub-optimal encoding algorithms exist. Texture compression, however, is an asymmetric application of compression, since decoding speed is essential while encoding speed is useful but not necessary.

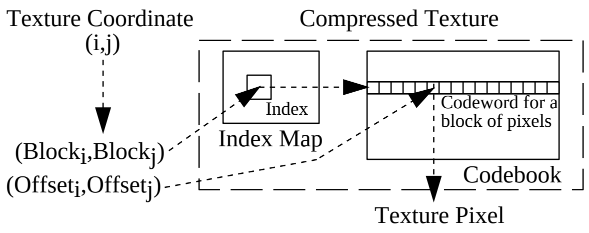

Figure 1: Accessing a pixel from a compressed texture.

3 Rendering

We have chosen VQ as our texture compression algorithm because it addresses all of the issues presented in the previous section, and in particular it supports fast decompression. When using VQ, we consider a texture as a set of pixel blocks. VQ attempts to characterize this set of blocks by a smaller set of representative blocks called a *codebook*. A lossy–compressed version of the original image is represented as a set of indices into this codebook, with one index per block of pixels. This set of indices is called the *index map*. The texture can be decompressed by looking up each block of pixels in the codebook via its index. In [9], such an indexed-lookup technique is used to page in uncompressed textures from disk on demand. Color quantization algorithms such as the Median Cut Algorithm[6] use this same indexed-lookup technique for representing 24 bit images with 8 bits per pixel. A method for compressing 3D volumes using VQ and a fast technique for volume rendering directly from this compressed format is presented in [8]. While we will describe exactly how to encode textures using VQ in section 4, we first show how to render directly from VQ compressed textures.

The rendering algorithm is outlined in the following algorithm. Assuming that the (s, t) texture coordinate has already been converted to an integral (i, j) location within the texture, we:

1. Determine in which block B pixel (i, j) lies, and the offset of pixel (i, j) within that block.

2. Lookup the index associated with block B to determine the corresponding codeword in the codebook.

3. Lookup the pixel (i, j) within this codeword block.

The three step process is shown pictorially in figure 1. Step 1 is easily implemented with fast bit shift and logical operators when texture and block sizes are a power of two. To achieve the highest compression rate we store the index map in a packed representation. Therefore, in step 2, two memory accesses and a few simple bit operations may be required if the size of each index is such that the indices do not fall on word boundaries. However, for word aligned indices, the appropriate index can be determined with a single lookup.

4 Encoding

The most critical part of encoding a texture using VQ is designing the codebook. The Generalized Lloyd Algorithm (GLA)[4] is one technique for generating a codebook. It is an iterative clustering algorithm which yields a locally optimal codebook for a given set of pixel blocks, called the *training set* of vectors.We generally use all the blocks in the original texture as our training set. The algorithm begins by selecting a set of potential codewords from the training set and then iterates on the following steps. Each training vector is grouped with the nearest codeword, based on some distortion measure such as Euclidean distance. The centroids of the groups are chosen as the new set of codewords, and the process repeats until the set of codewords converges.

Although this “Full Search” approach is locally optimal, generating the codebook is computationally expensive. A faster technique for producing the codebook is Tree Structured VQ[4]. This approach designs the codebook recursively, organizing the codebook as a binary tree. The first step is to find the centroid of the set of training vectors, which becomes the root level codeword. To find the children of this root, the centroid and a perturbed centroid are chosen as initial child codewords. The GLA is then used to converge on the locally optimal codewords for this first level in the tree. The training vectors are split into two groups based on these locally optimal codewords and the algorithm recurses on each of these subtrees. Note that once the child codewords have been chosen, the training vectors are permanently grouped with the nearest codeword. Since codewords cannot jump across subtrees, this approach is not guaranteed to produce the locally optimal codebook, but is substantially faster than Full Search VQ [4].

Once the codebook has been generated we encode a texture by mapping each block of pixels to the nearest codeword. With Full Search VQ we exhaustively search for the nearest codeword. With Tree Structured VQ, we can traverse the codebook tree always taking the path with the closest codeword. Thus, both codebook generation and texture encoding are faster with Tree Structured VQ than with Full Search VQ. We use Tree Structured VQ for quick experimentation, and Full Search VQ for generating final codebooks.

5 Texture Encoding Tradeoffs

In generating the VQ encoding for a texture we have control over several parameters that can be used to tradeoff compression rate for the quality of the compressed texture. This tradeoff must be considered carefully when encoding textures for a given scene. In this section we describe some of the parameters in VQ encoding that affect this tradeoff.

We have many choices on how to partition the image data into training vectors when designing a VQ codebook. The size of a vector is dictated by the dimensions of the block of pixels being coded and the number of color channels used to define the color of each pixel. We can either design a codebook for each color channel separately, or treat components of a color as a single value and code them together. We use the latter approach, which results in a higher compression rate since only one codebook and index map is used, instead of one codebook and index map per color channel.

The size of the codebook influences the compression rate in two ways. A larger codebook will lower the compression rate by increasing the size of the compressed representation. A larger codebook also means that the indices into the codebook will require more bits, increasing the size of the index map. However, a larger codebook will contain more of the representative blocks giving us better quality compressed textures. The size of the pixel block used in texture encoding also has a large effect on the overall compression rate. For example, if we use 4×4 blocks when encoding a texture, each RGB pixel block contains $4 \times 4 \times 3 = 48$ bytes. With a 256 entry codebook, we use a 1 byte index to represent each pixel block, yielding a base compression rate of $48 : 1$. The overall compression rate is reduced from the base rate by the storage requirements of the codebook. For example, when encoding a 512×512 image using a 256 entry codebook, the overall compression rate falls to $27.4 : 1$. With 2×2 blocks the base compression rate is only $12 : 1$. However, there are fewer possible 2×2 blocks than 4×4 blocks, and therefore, 2×2 codebooks produce better quality compressed textures than 4×4 codebooks. Similarly we could use larger blocksizes to gain higher compression rates for worse quality.

To achieve additional compression, we can encode three channel RGB textures in the 4:1:1 YUV format, commonly used in video standards. This format stores the color channels U and V decimated by a factor 2 both horizontally and vertically, and the luminance channel Y at full resolution. Converting from this YUV representa-

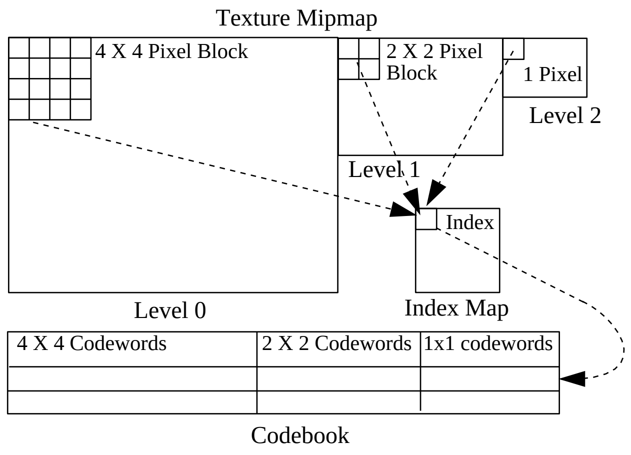

Figure 2: The encoding of three consecutive levels of a mipmap.

tion back to the RGB representation used by most renderers requires only multiplication by a constant 3x3 matrix, which is easily implemented in hardware.

We can also increase the compression rate by amortizing the cost of a larger codebook over several textures. We train a codebook on multiple textures to determine one codebook for all of them. A single codebook may be enough to characterize all of the textures in a scene. If one codebook results in too many compression artifacts, we can group the textures and design one codebook per group.

6 Mipmapping

An effective way to resample textures is by using a mipmap[11]. A mipmap stores a texture as an image pyramid, and is designed to allow efficient filtering of a texture. Each mipmap level stores a filtered version of the texture corresponding to a particular image pixel to texture pixel ratio. To apply a mipmapped texture, we compute texture coordinates (s, t) and an approximation to d, the image pixel to texture pixel ratio. Because these values are not integral in most cases, trilinear interpolation is used to determine the texture value.

One way to compress mipmaps would be to compress each mipmap level individually. However, we can take advantage of the correlation between successive layers of a mipmap by encoding several levels at once, generating one codebook as well as one index map for the group of levels. This allows us to gain higher compression rates in compressing mipmaps.

To compress a mipmap, we begin by compressing the original texture using 4×4 blocks. Given the codebook for this first mipmap level, we can form a codebook for the next level by averaging each 4×4 codeword down to a 2×2 codeword. Similarly, a codebook for the third level of the mipmap can be formed by averaging these 2×2 codewords down to single pixel codewords. Instead of storing three codebooks, we can combine them into one codebook containing extended codewords. Each extended codeword is formed by concatenating the 4×4 codeword with the corresponding subsampled 2×2 and 1×1 codewords. Note that the index maps for the three levels are identical, and therefore can be shared. The encoding of three mipmap levels is shown pictorially in figure 2. The full mipmap can be formed by repeating this process for every group of three levels, creating a separate index map and codebook for every group of three. In practice, once the size of a mipmap level is less than 32×32 pixels, we store it uncompressed.

For the trilinear interpolation, each (s, t, d) mipmap texture coordinate is converted to the eight nearest integral (i, j, L) mipmap locations in two adjacent levels of the mipmap. To access a pixel in a given level L we first determine the group of levels to which L belongs. The texture coordinates are then used to look up the index into the codebook from the index map for that group of levels. The level number determines in which part of the extended codeword the desired pixel is stored. Finally, the block offsets of the pixel are used to address the desired pixel.

Trial		Overall Rate	MSE
RGB Files:	All: 4096, 4×4, RGB	11.8:1	12.12
RGB Files:	All: 4096, 4×4, YUV	15.3:1	16.17
RGB Files:	Signs, Roads: 256, 2×2, YUV Other: 4096, 4×4, YUV	12:7:1	11.27
RGB Files:	Signs: 256, 2×2, YUV Roads, Other: 256, 4×4, YUV	24.5:1	16.19

Table 3: For each of these trials, one channel textures were compressed using 256, 2×2 codewords, and RGBA textures were compressed using 256, 2×2 codewords.

The drawback of our mipmap encoding approach is that the quality of the compressed level 1 and level 2 mipmaps can not be any better than the quality of the compressed level 0 mipmap, even though smaller sized codewords are used for levels 1 and 2. Interpolative VQ[5] [4] is an alternative technique for compressing image pyramids such as mipmaps. The general approach is to perform VQ on a subsampled image, decompress the image, interpolate it up to the next larger image size and VQ the difference between the interpolated image and the original. The benefits of this approach are that it takes advantage of the correlation between successive levels in the mipmap during encoding, and unlike the scheme we propose, the quality of each compressed mipmap level is somewhat independent of the quality of other mipmap levels. However, Interpolative VQ also requires a separate codebook per mipmap level, and retrieving a pixel from a particular mipmap level requires building up the the pixel value from the most subsampled mipmap level up to the desired level, accessing a codeword from each mipmap level in between. Although we have not performed direct comparisons to our encoding scheme, these two drawbacks make Interpolative VQ unattractive for fast texture mapping applications and we do not to use it in encoding mipmaps.

7 Results

We have evaluated our proposed VQ texture compression method by rendering texture mapped scenes using two different renderers: IRIS Performer and a custom software scan converter. We present the results for two scenes in this section. Since we cannot directly load our compressed format into Performer, we compress and decompress each texture in a preprocessing step that introduces compression errors into the textures. While we cannot directly compare rendering speed we can compare the visual quality of the rendered images. We also use a software scan converter to render textures directly from our compressed format. This allows us to compare both rendering times for, and the visual quality of, images rendered with and without compressed textures. We experiment with several of the VQ encoding parameters discussed in section 5 and report the compression rates for each of the scenes.

The Performer Town is a fully texture mapped virtual environment, containing 85 textures which require a total of 5 MB when uncompressed. Although most of the textures are three channel RGB textures, there are 14 one channel intensity textures and 7 four channel RGBA textures. We compress the intensity textures with a single codebook containing 256 codewords and a blocksize of 2×2. We generate a separate codebook for the RGBA textures again using 256 2×2 codewords. We vary several parameters in compressing the RGB textures and the overall compression results are given in table 3. The mean squared error (MSE) for a color channel is computed as the sum of squared differences between original frame pixels and corresponding pixels from the compressed texture frame, di-

Texturing used	Filter	Rate	Avg. MSE
Wood,Marble: 128, 4×4 YUV Top: 128, 2×2 YUV	Point	34.7:1	34.7
Wood,Marble: 256, 4×4 YUV Top: 256, 2×2 YUV	Point	28.3:1	19.1
all: 256, 2×2, YUV	Point	11.5:1	37.9
separate: 256, RGB	Mipmap	17.6:1	14.2
separate: 1024, RGB	Mipmap	8.0:1	13.3

Table 4: Compression rates and average MSE for 50 frames of the Topspin animation. As shown in plate 2, the scene contains three textures: Wood, Marble and Top.

Sampling	Textures used	Rendering Time	Increase
Point	Uncompressed	21.0 sec	
	VQ w/8 bit index	21.4 sec	1.7%
	VQ w/12 bit index	23.0 sec	9.6%
Mipmap	Uncompressed	65.4 sec	
	VQ w/8 bit index	69.5 sec	6.3%
	VQ w/12 bit index	78.3 sec	19.7%

Table 5: Rendering time for the Topspin animation for uncompressed and compressed textures using 8 and 12 bit indices. Twenty frames were rendered at 400×400 on a 132MHz MIPS R4600.

vided by the number of pixels in the frame. The MSE results we present are calculated for the frame shown in plate 1 and are averaged across the three RGB color channels.

The first two rows of table 3 present the results of using a single codebook across all the RGB textures. Using 4×4 blocks, certain textures such as signs, billboards and roads contain some noticeable artifacts. Based on this observation we separated the RGB textures into three groups: signs and billboards, roads, and all others, using a separate codebook for each group. This allows us to use smaller 2×2 blocks for the signs and roads, while using 4×4 blocks for the other textures. As shown in table 3, we can achieve slightly higher compression rates with a smaller average MSE using separate codebooks (see rows 1 and 3). Using separate codebooks, even at a compression rate of 24.5:1, the rendered scenes must be examined closely to see the artifacts in the textures, as shown in plate 1.

The Topspin animation contains three texture maps requiring 1.4 MB when uncompressed. We render this animation using a software scan converter. Some compression rates for this animation using point mapped and mipmapped textures are given in table 4. Note that the MSE numbers presented in this table are averaged across the 50 frames in the Topspin animation. We do not vary the codebook blocksize for mipmaps, since this is fixed by our three level codebook architecture. A separate compressed mipmap is generated for each of the three textures in the Topspin animation.

A frame from this animation rendered with point sampled textures is shown in plate 2, while the textures are shown in plate 3. The frame is almost free of artifacts, although there is some blockiness in the foreground Marble texture. The Wood and Marble textures in plate 3 have been cropped and enlarged to make the compression artifacts more visible.

The timing results for the Topspin animation are presented in table 5. Index maps with 12 bit indices take more time to decode than those with 8 bit indices because two lookups and bit manipulations are required to extract each index instead of a single lookup. Although these timing results indicate the computational overhead of rendering from VQ compressed textures in a software renderer, they do not represent a situation in which texture compression would be used. In a software renderer textures would only be compressed if the uncompressed textures surpass the main memory limit and cause the machine to swap. In such a situation, compression would alleviate the swapping, thereby drastically improving rendering times.

In a hardware implementation of this scheme, specialized addressing logic could be built to reduce the the penalty caused by irregular index sizes.

8 Conclusions

We have presented a method for rendering directly from VQ compressed texture maps. The advantage of using VQ over other compression schemes is that it addresses many of the issues involved in choosing a compression scheme for texture mapping. In particular, decompression is inexpensive. Even though VQ compression is lossy, we have been able to achieve compression rates of up to $35:1$ with few visible artifacts in the rendered images.

There are several directions in which this work may be extended. Designing codebooks currently requires some experimentation with the various VQ encoding parameters. While an automatic method for designing "optimal" codebooks would be useful, designing a measure of optimality is difficult. There has been some work on designing perceptual distortion measures to minimize such distortions in compressed images [2]. For texture mapping however, distortion in the rendered scene, not the compressed textures, must be minimized, so the distortion measure must use information about how the textures will be mapped into the scene. It may be possible to use a hint driven approach that allows the application designer to provide hints about characteristics like which textures are similar to one another, or which textures are more or less important to preserve perfectly. The VQ compression approach naturally extends to other classes of texture maps such as bump maps, displacement maps and environment maps. Each of these classes of textures has some unique filtering or access issues and although our preliminary results indicate that VQ compression works well for them, we are studying them in more detail.

Using VQ compressed textures in a rendering system is a viable method for reducing the memory overhead of texture mapping. Such compression is ideal for rendering systems that use specialized texture memory and aim for real-time performance. It will allow lower-end systems such as PCs and home game systems to achieve greater graphical realism through the use of more complex textures.

Acknowledgments

Homan Igehy and Gordon Stoll provided the scan converter used to generate timing results. The Performer Town database appears courtesy of Silicon Graphics and Paradigm Simulation.

References

[1] Kurt Akeley. RealityEngine graphics. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 109–116, August 1993.

[2] N. Chaddha, P. Chou, and T. Meng. Scalable compression based on tree structured vector quantization of perceptually weighted generic block, lapped and wavelet transforms. *IEEE International Conference on Image Processing*, October 1995.

[3] Lawrence French. Toy story. *Cinefantastique*, 27(2):36–37, 1995.

[4] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, 1991.

[5] H.-M. Hang and B. Haskell. Interpolative vector quantization of color images. In *TCOM*, pages 465–470, 1987.

[6] Paul S. Heckbert. Color image quantization for frame buffer display. In *Computer Graphics (SIGGRAPH '82 Proceedings)*, volume 16, pages 297–307, July 1982.

[7] Paul S. Heckbert. Survey of texture mapping. In M. Green, editor, *Proceedings of Graphics Interface '86*, pages 207–212, May 1986.

[8] P. Ning and L. Hesselink. Fast volume rendering of compressed data. In G. Nielson and D. Bergeron, editors, *Proc. Visualization '93*, pages 11–18, October 1993.

[9] Darwyn Peachey. Texture on demand. Technical report, Pixar, 1990.

[10] W.B. Pennebaker and J.L. Mitchell. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, 1993.

[11] Lance Williams. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17, pages 1–11, July 1983.

[12] L. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inform.Theory, Vol.IT-23*, (3), May 1977.