# Fast Resampling Using Vector Quantization

Patrick C. Teo* and Chase D. Garfinkle†

*Department of Computer Science
Stanford University, Stanford, CA 94305
†Silicon Graphics Computer Systems
Mountain View, CA 94043

**Abstract.** We present a fast resampling scheme using vector quantization. Our method differs from prior work applying vector quantization to speeding up image and volume processing in two essential aspects. First, our method uses blocks with overlapping rather than disjoint extents. Second, we present a means of trading off smaller block sizes for additional computation. These two innovations allow vector quantization to be used in performing a broader class of operations. We demonstrate the performance of our method in warping both images and volumes, and have also implemented a ray-traced volume renderer utilizing this technique. Experiments demonstrate a speed up of 2-3 times over conventional resampling with minimal errors.

## 1 Introduction

Resampling, the process of extracting values from gridded data, is a ubiquitous operation in image processing and computer graphics. Most applications which use data in the form of a stored image or volume must perform this operation many times, and computation time is often dominated by this cost. Such applications include many types of image and volume warping, filtering, texture mapping and volume rendering. In general, the locations at which data is to be sampled may be non-integral, or subpixel. Because there is no explicit representation of the data at these locations, resampling consists of interpolating the data at nearby grid points. The set of weights used to combine these points is known as the resampling kernel. Performing a single resampling operation consists of the following steps:

1. Load from memory all data which contribute to the computation of the interpolated value. For example, linear interpolation requires the nearest four samples in an image and eight in a volume.
2. Compute the weights for each of the samples. These weights are often the cartesian product of one-dimensional filters in each dimension of the data. For a linear interpolant, these functions are simple linear ramps
3. Multiply each sample value by the corresponding weight and sum to compute the interpolated value.

We can see that there are two fundamental costs in resampling: memory accesses to retrieve stored data, and floating-point operations to interpolate from these samples. In conventional resampling, these costs are proportional to the size of the resampling kernel. In this paper, we present a method based on Vector Quantization (VQ) which allows the resampling operation to be performed with fewer memory access and no run-time computation of the weights. The rest of the paper is organized as follows. In section 2, we present background on VQ and prior work using VQ in conjunction with image and volume processing operations. In section 3, we describe our technique for fast resampling. Section 4 gives the results of using our method in several applications. Finally, we conclude in section 5 with suggestions for extensions and future applications.

## 2 Background

*Vector Quantization.* Vector Quantization is a lossy data compression scheme, which can be understood as a generalization of scalar quantization. In scalar quantization, a range of possible values is mapped onto a smaller discrete set of representative values. This mapping operation might consist of an operation such as "rounding to the nearest multiple of four". In general, no restriction needs to be placed on the possible set of quantized values; they may be unevenly spaced, or even be non-integers. When this set is explicitly enumerated, it is commonly referred to as a *codebook* and its elements are called *codewords*. The process of choosing the codewords of a codebook is known as codebook design. Once the codebook has been designed, each input value is mapped to one of the codewords via some encoding rule (typically, the nearest neighbor rule).

Unlike scalar quantization, vector quantization quantizes sets of numbers (or *vectors*) as a group, rather than each value individually. As in scalar quantization, VQ codebook design involves producing a set of vectors which can be used to represent some collection of input vectors with the minimum possible error, or distortion. Various metrics can be used to measure distortion; squared error is the most common, and we have used it in this work. To design codebooks, we have used the pairwise nearest neighbor (PNN) algorithm[4], which operates by grouping the input vectors into clusters, and then choosing one representative vector for each cluster.

*Codebook Design.* The PNN algorithm works by first organizing the set of input vectors into a *k-d tree*[1], a data structure which spatially partitions data along axis-aligned planes to form a balanced binary tree. We form the tree by recursively subdividing the data until each leaf of the tree contains fewer than some fixed number of vectors. We choose the axis along which to subdivide each node to be the one in which the vectors have the maximum variance. We then choose the median location of the vectors along this axis as the plane with which to subdivide the node.

Once we have constructed the initial tree, the clustering proceeds as follows. We first consider each vector to be a cluster of size one, and then repeat the following process until the total number of clusters is reduced to the desired number of codewords.

1. Select from each leaf of the k-d tree a candidate pair of clusters such that combining them, and representing the aggregate by its centroid, will yield the smallest distortion over all pairs in that node.

2. For a fixed percentile of the candidate pairs, ranked by distortion, combine the clusters.
3. Rebalance the k-d tree.

Once the number of clusters is reduced to the number of desired codewords, the centroids of the remaining clusters are used as the codewords of the codebook. Algorithms that generate optimal codebooks are computationally expensive[5]. As a result, codebooks are commonly designed for use across a large number of data sets. This avoids the need to design a new codebook for each data set. While the PNN method is not guaranteed to generate the optimal codebook for a given set of input vectors, it generates good codebooks in practice, and is sufficiently fast that we are able to compute a new codebook for each data set.

*Prior Work.* A number of authors have previously explored the use of VQ to accelerate image or volume processing operations. The general technique has been to encode the data, perform the desired operation on the vectors of the codebook, and then use this processed codebook in the decoding stage. For example, in [2] and [3], the authors describe performing histogram equalization on VQ encoded images. In the case of global equalization, each codeword is equalized using a histogram for each image. Adaptive equalization is achieved by computing a number of versions of each codeword, each equalized for a different region of the image, and interpolating between these copies during decoding. Another interesting example is the use of VQ in volume rendering, presented in [6]. The author presents an orthographic ray-tracing volume renderer which operates in two stages. First, each codeword is individually rendered into a pixmap using conventional volume ray-tracing. The rendering of the entire volume is then produced by stepping from block to block within the volume along the projection direction and compositing the pixmaps for the codewords encountered.

Our method differs from prior work in two essential aspects. First, in prior work the vectors have encoded non-overlapping blocks in the input data. In this work, however, we allow the blocks to overlap. This makes a larger class of operations amenable to such a VQ approach. In our particular application, overlapping blocks are necessary to guarantee that all the data required to interpolate a sample is contained in a single block. Second, we present a means of trading off smaller block sizes for additional computation. Typically, the size of the block is determined by the spatial extent of the input region of the operation. Hence, when the input region is large, the size of the block may become prohibitively large for such a VQ scheme. By decomposing the operation appropriately, we can use smaller blocks at the cost of a moderate increase in computation.

## 3 Methods

### 3.1 Preprocessing

Our method involves three preprocessing steps: (1) designing the VQ codebook, (2) computing the extended codebook, and (3) encoding the input data. During codebook design, the input data is decomposed into overlapping blocks, each of which is usually the size of the resampling kernel. For example, when resampling image data with a bilinear interpolant, a $2 \times 2$ block is used (as shown in Figure 1). The use of overlapping blocks is a crucial difference from standard VQ

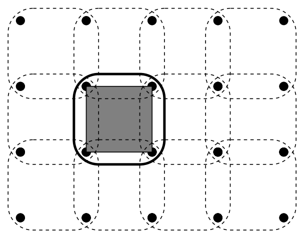

**Fig. 1.** Decomposition into overlapping blocks of an image which is to be resampled using a bilinear interpolant. Filled circles denote locations of image pixels. Resampling of the image at any location within the shaded region will be computed from the extended VQ codebook entry of the enclosing block (see Section 3.2.

methods. If the input data were encoded as disjoint blocks, the computation of a new sample value might require samples which reside in several different blocks. By using overlapping blocks, we can guarantee that all the data necessary to compute a sample will reside in a single block.

A small set of representatives is then derived from the blocked data to form the initial VQ codebook (as described in Section 2). Next, the extended codebook is generated by resampling each codeword in the initial VQ codebook at some finite number of subpixel locations. Lastly, each overlapping block in the input data is encoded with the index of its best representative from the codebook.

The extended VQ codebook contains the same number of entries as the original VQ codebook. However, unlike the entries in the original codebook which are blocks of values from the original data, each entry in the extended codebook is a small table of resampled values computed at a fixed number of subpixel locations within the block. Hence, we can index these resampled values by the concatenation of the original codebook index and an index designating the discretized subpixel location.

When resampling image data with a bilinear interpolant, for example, each entry in the original codebook is a $2 \times 2$ block of pixel values. If the subpixel displacements are discretized onto a $4 \times 4$ grid, then each entry of the extended codebook is made up of 16 values computed by resampling the corresponding block in the original codebook at each subpixel location on the grid. Typically, codebooks of 256 entries are used for 8-bit gray-level images. As a result, the index into this extended codebook would be 12 bits wide, consisting of 8 bits for the original codebook index and 4 bits to encode the discrete subpixel location. Figure 2 shows a typical entry in an original VQ codebook along with its corresponding extended VQ codebook entry.

### 3.2 Resampling using the Extended VQ Codebook

After preprocessing, the input data can be efficiently resampled such that only two memory accesses are required, one to the encoded data, and one to the codebook. First, the integral grid location is computed and the codebook index

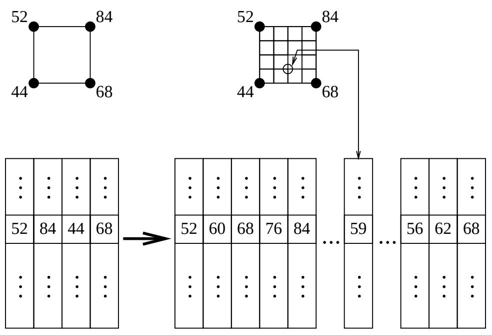

**Fig. 2.** Original and extended VQ codebook entries for image resampling using a bilinear interpolant. (a) One entry in the original VQ codebook. This entry corresponds to a $2 \times 2$ block of image pixels. (b) Corresponding entry in the extended VQ codebook. This entry contains a $4 \times 4$ table of image pixel values resampled at the corresponding subpixel locations.

for this location is retrieved from the encoded input data. For example, if we want to draw a sample at location $(53.3, 28.7)$ in an image, we would retrieve the codebook entry whose index is stored at location $(53, 28)$. Second, the fractional displacement of the new location from this grid point is used to retrieve the precomputed, resampled value from the codebook. If we have resampled the codebook on a $4 \times 4$ grid, we would quantize the offset in our example to $(\frac{1}{4}, \frac{3}{4})$, and so retrieve the value at position $[1, 3]$ within the codeword.

In comparison to these two memory accesses, bilinear interpolation of scalar image data requires four memory accesses and three linear interpolations. Trilinear interpolation of scalar volume data requires eight memory accesses and seven linear interpolations.

### 3.3 Trading off block size for computation.

Using the method described so far, interpolants with larger kernels require larger block sizes. For example, a bicubic interpolant for images would require blocks that are $4 \times 4$ pixels wide and a tricubic interpolant for volumes would require blocks that are $4 \times 4 \times 4$ in size. In order to keep the average quantization error low, larger block sizes must be accompanied by larger codebooks. Unfortunately, VQ becomes unmanageable as codebooks become too large. On the other hand, if the codebook size is kept fixed, the average quantization error introduced increases with the use of larger blocks.

Our solution is to adopt a hybrid method in which smaller block sizes are traded off for additional computation. For interpolations that can be decomposed into a sequence of operations with smaller supports, the size of each block only needs to be the size of these smaller support regions. For example, when resampling an image with a bicubic interpolant, a $4 \times 1$ pixel block could be used. Resampling now requires looking up the codebook entries associated with four contiguous blocks and combining them using a one-dimensional cubic interpolant. Section 4 shows that despite the added computation, the hybrid method still outperforms the conventional method by more than a factor of two.

## 4 Results

### 4.1 Performance

We assess the improvement afforded by our algorithm in resampling image and volume data. For each of these data sets, we compare the performance of our method against conventional resampling when used with linear and cubic interpolants. Performance is measured by the time required to warp and resample the entire data set, including the time required to compute the new sample locations. We also compute the mean squared error between the results generated by our method and by conventional resampling. Implementations of both our method and the conventional method were optimized for speed. All running times reported are for an implementation on a Sun Microsystems Sparc10 workstation with 32 Mb of main memory.

*Image Resampling.* Figure 3 shows two versions of a $512 \times 512$, 8-bit gray-level image, resampled using a bilinear interpolant. The image on the left was resampled using the conventional method while the image on the right was resampled with our method. The codebook contained 512 entries, and subpixel locations were discretized onto an $8 \times 8$ grid, yielding a 32Kb extended codebook.

Table 1 reports the average times taken by our method and the conventional method to rotate the image over a range of $2\pi$ radians in 256 uniform steps, using both bilinear and bicubic interpolants. Note that the last four rows of the table report results for the hybrid method.

*Volume Resampling.* Table 2 reports the times taken by our method and the conventional method to perform an affine warp on a $200 \times 200 \times 200$, 8-bit volume, using trilinear interpolation. Note again that the last two rows of the table report results using the hybrid method.

### 4.2 Application (Volume Rendering)

Figure 4 shows ray-traced volume renderings of a $200 \times 200 \times 200$, 8-bit volume data set. Trilinear interpolation was used to resample the volume. The image

| | Block Size | Codebook Size | VQ Resampling (sec) | Conventional (sec) | MSE |
|---|---|---|---|---|---|
| Bilinear | $2 \times 2$ | 256 | 0.62 | 1.45 | 7.30 |
| Bilinear | $2 \times 2$ | 512 | 0.63 | 1.45 | 5.37 |
| Bicubic | $4 \times 4$ | 2048 | 0.72 | 2.48 | 18.01 |
| Bicubic | $4 \times 4$ | 4096 | 0.73 | 2.48 | 14.65 |
| Bicubic | $4 \times 2$ | 512 | 0.89 | 2.48 | 17.32 |
| Bicubic | $4 \times 2$ | 1024 | 0.95 | 2.48 | 14.42 |
| Bicubic | $4 \times 1$ | 256 | 1.00 | 2.48 | 8.14 |
| Bicubic | $4 \times 1$ | 512 | 1.01 | 2.48 | 5.57 |

**Table 1.** Times required to resample a $512 \times 512$ image of 8-bit data and their associated mean squared errors (MSE).

|  | Block Size | Codebook Size | VQ Resampling (sec) | Conventional (sec) | MSE |
|---|---|---|---|---|---|
| Trilinear | $2 \times 2 \times 2$ | 1024 | 35 | 76 | 6.75 |
| Trilinear | $2 \times 2 \times 2$ | 2048 | 35 | 76 | 5.13 |
| Trilinear | $2 \times 2 \times 1$ | 256 | 41 | 76 | 4.04 |
| Trilinear | $2 \times 2 \times 1$ | 512 | 41 | 76 | 2.66 |

**Table 2.** Times required to resample a $200 \times 200 \times 200$ volume of 8-bit data and their associated mean squared errors (MSE).

on the left was rendered using the conventional resampling method. The image on the right was rendered using our method with a codebook size of 2048 and a block size of $2 \times 2 \times 2$. In rendering the image on the right, the codebook was used to precompute gradients needed to shade the volume. The image rendered using the conventional resampling method took 200 seconds while the image rendered using our method took 94 seconds on a 150 MHz Silicon Graphics Indigo2 with 64 Mb of main memory.

## 5 Conclusions

We have presented a fast resampling scheme using vector quantization. Our method differs from prior work in two essential aspects. First, our method uses blocks with overlapping extents. Second, we present a means of trading off smaller block sizes for additional computation. Experiments using our method to resample images and volumes demonstrate a speed up of 2-3 times over conventional resampling with only small errors. In general, the use of overlapping blocks allows vector quantization to be applied in image/volume processing applications where disjoint blocks do not provide sufficient information. Our fast resampling method can also be used on non-rectangular grids.

## References

1. J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, December 1979.
2. P. Cosman, K. Oehler, E. Riskin, and R. Gray. Combined vector quantization and adaptive histogram equalization. In *SPIE Proc. Medical Imaging VI*, 1992.
3. P. Cosman, K. Oehler, E. Riskin, and R. Gray. Combining vector quantization and histogram equalization. *Information Processing Management*, 28(6):681–686, 1992.
4. W. Equitz. A new vector quantization clustering algorithm. *IEEE Trans. on Acoustics, Speech and Signal Processing*, 37(10):1568–1575, 1989.
5. Y. Linde, A. Buzo, and R. Gray. An algorithm for vector quantizer design. *IEEE Trans. on Communications*, 28:84–95, 1980.
6. P. Ning. Applications of data compression to 3-d scalar field visualization. Technical report, Stanford University, 1993. PhD Dissertation.

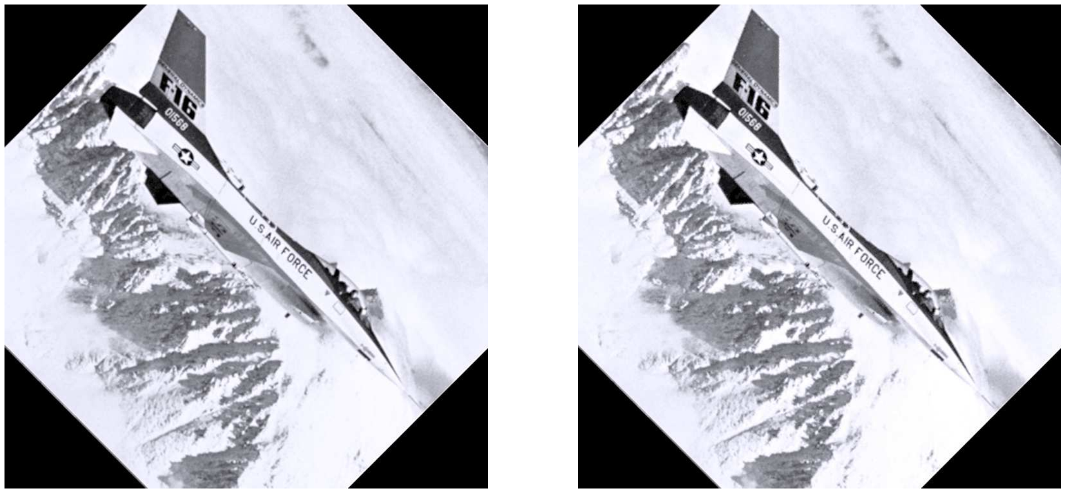

**Fig. 3.** Both images are bilinearly resampled versions of the original $512 \times 512$, 8-bit gray-level image. The image on the left was resampled using the conventional method while the image on the right was resampled using our method.

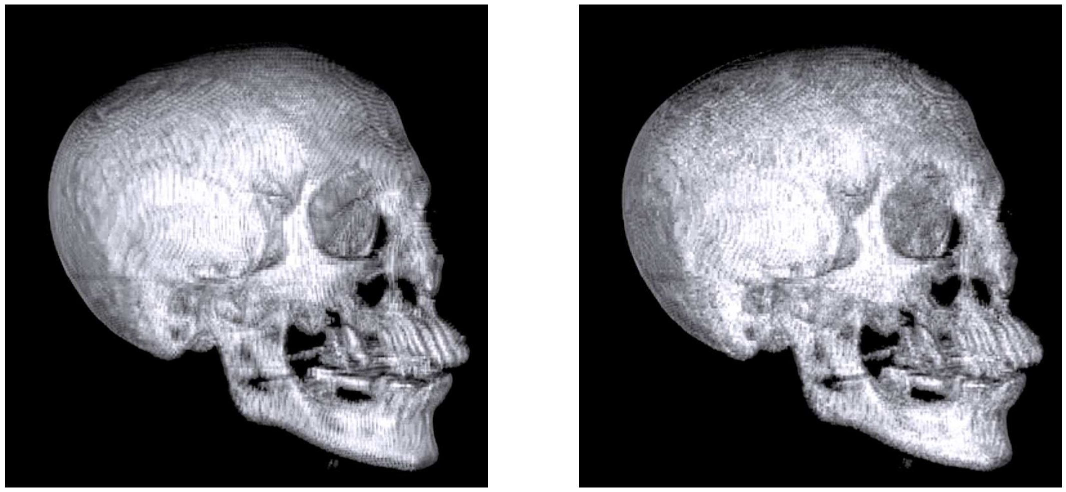

**Fig. 4.** Both images are ray-traced volume renderings of a $200 \times 200 \times 200$, 8-bit volume data set. Trilinear interpolation was used to resample the volume. The image on the left was rendered using the conventional resampling method. The image on the right was rendered using our method with a codebook size of 2048 and a block size of $2 \times 2 \times 2$.