Fast and Compact Volume Rendering in the Compressed Transform Domain [*]

Shenfeng Chen and John H. Reif
Duke University

August 30, 2003

Abstract

Potentially, data compression techniques may have a broad impact in computing not only by decreasing storage and communication costs, but also by speeding up computation. For many image processing applications, the use of data compression is so pervasive that we can assume the inputs and outputs are in a compressed domain, and it is intriguing to consider doing computations on the data entirely in the compressed domain. In this paper, we speed up processing by doing computations, including dot product and convolution on vectors and arrays, in a compressed transform domain. To do this, we make use of algebraic techniques for evaluation and interpolation of sparse polynomials. We illustrate the basic methodology by applying these techniques to image processing problems, and in particular to speed up the well known splatting algorithm for volume rendering. The splatting algorithm is one of the most efficient of existing high quality volume rendering algorithms; it takes as input three dimensional volume sample data of size N^3 and outputs an $N \times N$ image in $O(N^3 f)$ time, where f is a parameter known as footprint size (which often is hundreds of pixels in practice). Assuming that the original sample data and the resulting image are stored in the transform domain and can be lossily compressed by a factor ρ with small error, we show that the rendering of the image can be done entirely in the compressed transform domain in decreased time $O(\rho N^3 \log N)$. Hence we obtain a significant speedup over the splatting algorithm when $f \gg \rho \log N$. The same methodology of computing in compressed transform domain can be applied to speed up other algorithms for image processing problems.

[*]Surface address: Department of Computer Science, Duke University, Durham, NC 27708. Email addresses: reif@cs.duke.edu and chen@cs.duke.edu. Supported by NSF Grant NSF-IRI-91-00681 and Army Research Office contract DAAH-04-96-1-0448.

1 Introduction

Computing in Transform Domain. There has been extensive research on lossless and lossy image compression to represent image with less data in order to save storage space and reduce the time and cost of transmission [9, 14]. Various transform compression techniques, such as Discrete Cosine Transform, Fast Fourier Transform, wavelet transform, Huffman coding, have been developed to design efficient image compression algorithms. For example, the well known image compression standard - JPEG, uses the Discrete Cosine Transform to encode the image by dividing the image into small blocks and encoding each block individually. It is also well known that many image processing tasks, such as filtering and convolutions, can be speed up by computing directly in the transform domain instead of spectral domain [5, 13].

Computing in Compressed Transform Domain. In this paper, we apply the novel techniques of speeding up running time of an algorithm by representing data and performing computation in *compressed* transform domain. However, there are a number of elements of our work that draw on prior work. The representation we use for compression of the transformed images is similar to the standard image encoding methods used in image compression algorithms [14]. This is an advantage, in that we can expect to obtain similar image compression and corresponding degradation as would be obtained by similar conventional transform image techniques. There has been prior work done which reduces computation of an algorithm by taking into account of certain statistical properties of the input data, such as the bounded entropy (see [7]). Furthermore, there has been research on computing directly in the *compressed* domain, e.g., [1] gave an string matching algorithm that finds the pattern string in the compressed text without decompressing the text. There has been extensive research on computation in transform domain but with the exception of searching algorithms, computing in a compressed transform domain is a novel idea.

We assume that the inputs are vectors or arrays already stored in the compressed transform domain. Similarly, we assume that the outputs in the compressed transform domain, and we analyze the computational cost of our algorithm accordingly. The operations we perform on vectors and arrays include addition (which is easily done either in the uncompressed domain or the compressed domain), as well as convolution and dot product. These operations are common in image processing applications, where for example, we may perform sequences of filtering and density masking on the image domain. In prior practice one maps to and from the transform domain when applying the convolution in combination with dot product. Since the compressed transform domain is assumed to be small, ideally we would like to instead do all our computations in the compressed transform domain. We note that certain operations, such as convolution, can be performed easily if the inputs are mapped to the transform domain. On the other hand, other operations such as dot product can be done easily on the inputs in the compressed domain, but not so straightforwardly if the inputs are mapped to the transform domain (since the naive method for performing dot product needs to convert the vectors to the original domain and convert the result of dot product in the original domain back into the transform domain, which requires $O(n \log n)$ time in one dimension and $O(n^2 \log n)$ time in two dimension). Instead, we perform these non-trivial operations such as the dot product by algorithm techniques using sparse polynomials. For problems in the areas such as image and speech processing where the input data is known to have high lossy compression ratio with low L_2 error, our idea of computing in compressed transform domain can be applied to greatly reduce computational cost. In particular, we apply this methodology to the problem of designing fast and compact volume rendering algorithms in compressed transform space. Furthermore, we expect that this novel idea of reducing computation by computing in compressed transform domain can be applied to other image processing problems requiring sequences of filtering and density masking in the image domain.

Application to Volume Rendering. A *volume rendering* algorithm takes as input three-dimensional arrays of voxels (3D pixels) of size $N \times N \times N$, giving the discrete input sample volume to be rendered, and outputs an $N \times N$ image rendering. Volume rendering is an ideal application of our techniques: (i) the inputs are generally very large (N often can be over $1,000$, so the volume input size is often a number of megabytes), so it is advantageous to have compressed the inputs, (ii) the volume may be viewed at many viewing angles, creating a multiplicity of output images so it is also often advantageous to also compress the outputs for subsequent viewing, and furthermore (iii) the operations required by volume rendering algorithms are similar to many other image processing applications, and in particular include operations which we may speed up by computing in the compressed transform domain.

The "splatting" algorithm by Westover [17] is perhaps the most efficient existing high quality volume rendering algorithms, and runs faster than ray casting methods by use of an approximation technique known as footprints.

We present an improved volume rendering algorithm which reduces the computation of the "splatting" algorithm by computing in compressed transform domain. We will describe how to render the image efficiently by computing in compressed transform domain. Assuming that the original sample data and the resulting image are stored in the transform domain, we show that the rendering of the image can be performed in $O(\rho N^3 \log N)$ where N^3 is the size of the three dimensional sample data and ρ is the compression factor, compared to the $O(N^3 f)$ running time of the original splatting algorithm where f is the size of footprint. Our algorithm is more efficient when $f \gg \rho \log N$.

Organization of This Paper. This paper is organized as follows. In Section 2, we will describe the general problem of computing in compressed transform domain. In Section 3, we describe some known volume rendering algorithms, including the splatting algorithm. In Section 4, apply our method to the splatting volume rendering algorithm and give a new algorithm which performs computation in the compressed transform domain. We also present the performance analysis of our algorithm and discuss the advantages of our algorithm. We summarize our results in Section 5.

2 Computing in Compressed Transform Domain

We will use *Discrete Fourier Transform* (DFT) as the transform for the presentation of this paper. Let $\omega_n = \exp(2\pi\sqrt{-1}/n)$ be the nth root of unity over the complex numbers. Let $\vec{a}$ be an n-vector such that $\vec{a} = (a_0, a_1, ..., a_{n-1})$. We define the vector $\vec{y} = (y_0, y_1, ..., y_{n-1})$, where $y_k = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$, to be the DFT of vector $\vec{a}$. We also write $\vec{y} = \mathrm{DFT}_n(\vec{a})$. As is well known, the DFT_n is the vector of values $(p(\omega_n^0), p(\omega_n^1), \ldots, p(\omega_n^{n-1}))$ of the polynomial $p(x) = \sum_{j=0}^{n-1} a_j x^j$ with coefficients $\vec{a}$. Using the Fast Fourier Transform algorithm. which takes advantage of the special properties of the complex roots of unity, the DFT and inverse DFT of vector $\vec{a}$ of length n can be computed in $O(n \log n)$ time. (In practice where the input data set will most likely contain real data, an alternative transform such as *Cosine Transform* or *Fast Hartley Transform* (FHT) may be more advantageous.) For example, the FHT is defined as follows:

$$H(k_x, k_y, k_z) \quad = \quad \frac{1}{N^3} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \sum_{z=0}^{N-1} f(x, y, z)[\cos(2\pi \frac{xk_x + yk_y + zk_z}{N})$$

$$+\sin(2\pi\frac{xk_x + yk_y + zk_z}{N})]. \tag{1}$$

We will use Discrete Fourier Transform in the discussion of this paper while the other transforms can be applied as well. We will first show the standard operations in transform domain and then describe how to execute these operations in the compressed transform domain.

Computing in Transform Domain. Without loss of generality, we consider the problem of computing in transform domain in one dimension. These operations are easily extendible to higher dimensions. Let $\mathcal{U}_1$ denote the untransformed original one-dimensional domain and let $\mathcal{T}_1$ denote the one-dimensional transform domain. The operations we consider are vector operations such as addition of vectors, dot product of vectors and convolution of vectors in $\mathcal{U}_1$. Let $\vec{u}$ and $\vec{v}$ be vectors of length n and the result of the operation be vector $\vec{r}$. We define these operations as follows: **(i) addition:** "+" $\vec{u}+\vec{v}=\vec{r}$, $r_i = u_i + v_i$, **(ii) dot product:** "·" $\vec{u}\cdot\vec{v}=\vec{r}$, $r_i = u_i v_i$, and **(iii) convolution:** "$\star$" $\vec{u}\star\vec{v}=\vec{r}$, $r_i = \sum_{k=0}^{i} u_k v_{i-k}$.

The addition and dot product can be performed easily in $O(n)$ time in original domain. By the *Convolution Theorem*, the convolution of two $n-$vectors can be computed by mapping the vectors to the transform domain using the DFT_{2n}, performing dot product on the transformed DFT_{2n} vectors and then computing the inverse Discrete Fourier Transform (DFT_{2n}^{-1}) of the resulting vector, both taking $O(n\log n)$ time. This is routine for signal processing applications. On the other hand, the convolution of two vectors given in the transform domain $\mathcal{T}_1$ can be computed by computing an inverse transform (e.g. an inverse DFT) of the two vectors, computing a dot product of the resulting vectors in the original domain $\mathcal{U}_1$, and then converting the result back into DFT in $\mathcal{T}_1$. In other words, the convolution operation and dot product operation are dual problems in original and transform domain. But note that conversion to and from these domains $\mathcal{U}_1, \mathcal{T}_1$ appears to be required if we wish to do combinations of dot product and convolution operations.

The Compressed Transform Domain. To create a compressed transform domain $\mathcal{T}_1$, we need to drop data without introducing large error. A technique known to provide good performance in practice for many classes of images and speech data, is to remove high frequency terms in the Discrete Fourier Transform of the vector which are close to zero. Specifically, let $\vec{v} = (v_0, v_1, ..., v_{n-1})$ be a n-vector which consists of the coefficients of the polynomial $q(x) = \sum_{i=0}^{n-1} v_i x^i$. We encode $\vec{v}$ by Discrete Fourier Transform $s_0, s_1, ..., s_{n-1}$ where $s_i = \sum_{j=0}^{n-1} x_j \omega^{ij}$. We keep k low frequency terms,

$s_0, s_1, ...s_{k-1}$. In the remaining terms, we keep s terms of highest magnitude. Let $n' = k + s \ll n$. The total storage space needed is ρn where $\rho = \frac{n'+s}{n}$ since we need to store the position indices of s terms of the highest magnitude as well. We consider the problem how to compute the addition, dot product and convolution of two vectors in the original domain $\mathcal{U}_1$ by using their corresponding compressed vectors in the compressed transform domain $\mathcal{T}_1$. The result is stored in the compressed transform domain $\mathcal{T}_1$. (Our compression scheme is similar to many one dimensional transform compression schemes in that only the high frequency (sparse) terms which contain non-vital information are dropped. The similarity is advantageous, since it indicates that we may obtain compression ratio comparable to to that of these conventional transform compression methods. The main difference between our compression scheme and typical one dimensional transform compression method is that such methods may apply the transform only on blocks of consecutive subsequences of some bounded size, in part to limit computational costs. In practice, to further decrease our computational costs, we can also apply this decomposition method.)

A *sparse polynomial* is a polynomial of degree n which has s non-zero terms where $s \ll n$. The compressed transform of a vector $\vec{a}$ in the transform domain has a corresponding sparse polynomial $p(x) = \sum_{i=0}^{n-1} a_i x^i$ whose coefficients form the same vector. In [2, 3, 10], it is shown that the evaluation and interpolation of a sparse polynomial with degree n and s terms at s arbitrary points can be accomplished in $O(s \log^2 s + \log n)$ time. Furthermore, if the evaluation or interpolation points are the s roots of unity (or more generally, at s chirp points ζ^i $(i = 0, ..., s-1)$, for some fixed complex number ζ), the cost is $O(s \log s + \log n)$.

Operations using Compressed Transform Domain. We consider the problem how to execute operations such as addition, convolution and dot product on vectors in the original domain $\mathcal{U}_1$ by using their corresponding representations in the compressed transform domain $\mathcal{T}_1$. Without loss of generality, we consider the operations on two n-vectors $\vec{u}$ and $\vec{v}$ in one dimension (we later will extend these operations to two dimensional arrays) using their corresponding vectors $\vec{a}$ and $\vec{b}$ in the compressed transform domain $\mathcal{T}_1$ which are encoded using the compression method described above. Let k be the number of low order coefficients in $\vec{a}$ and $\vec{b}$ which we store (that is, we always store low order coefficients $a_0, a_1, \ldots, a_{k-1}$ and $b_0, b_1, \ldots, b_{k-1}$ as dense vectors), and let s be the number of additional sparse high order terms in $\vec{a}$ and $\vec{b}$ which we store (that is, we also store the lists of coefficients $a_{i_0}, a_{i_1}, \ldots, a_{i_{s-1}}$ and $b_{j_0}, b_{j_1}, \ldots, b_{j_{s-1}}$ of highest magnitude). Thus the *un-*

compressed size is the length n of the vector in the untransformed domain $\mathcal{U}_1$, and the *compressed size* is the number n' of non-zero coefficients of the vectors in the compressed transform domain $\mathcal{T}_1$. We assume that the input and output vectors are both stored in the compressed transform domain $\mathcal{T}_1$ and the computational cost is analyzed accordingly, in terms of n', where possible.

Addition. The addition of two vectors $\vec{u}$ and $\vec{v}$ in the untransformed domain $\mathcal{U}_1$ can be performed by adding the corresponding terms of $\vec{a}$ and $\vec{b}$ in $\mathcal{T}_1$. Note that in the worst case, the number of non-zero terms in the resulting vector $\vec{z}$ may become $k + 2s$ where k is the number of dense frequency terms and s is the number of sparse terms in vector $\vec{a}$ or $\vec{a}$. We keep the k dense terms and choose s sparse terms of highest magnitude out of the possible $2n$ sparse terms. Thus the addition in $\mathcal{U}_1$ will be computed in the compressed transform domain $\mathcal{T}_1$ which takes $O(n')$ time. We denote the approximated computation of addition by $ApproxAdd()$.

Convolution. Given vectors $\vec{u}$ and $\vec{v}$ in untransformed domain $\mathcal{U}_1$, recall that their convolution can be computed by an inverse transform of the dot product of their compressed transform vectors $\vec{a}$ and $\vec{b}$ in the transform domain $\mathcal{T}_1$. We first perform pointwise multiplication of the corresponding terms of $\vec{a}$ and $\vec{b}$ in $\mathcal{T}_1$. Let the resulting vector be $\vec{r}$ of length n such that $r_i = a_i b_i$. The number of non-zero elements in $\vec{r}$ is equal to or less than n'. Thus the convolution in $\mathcal{U}_1$ will be approximately computed in compressed transform domain $\mathcal{T}_1$ in $O(n')$ time. We denote the approximated computation of convolution by $ApproxConv()$.

Dot Product The dot product of two given $n-$vectors $\vec{u}$ and $\vec{v}$ in the untransformed domain $\mathcal{U}_1$ is a vector $(u_0 + v_0, u_1 + v_1, \ldots, un - 1 + v_{n-1}$ whose ith element is the sum of the i elements of u and v.

To compute the dot product of two vectors $\vec{u}$ and $\vec{v}$, we need to compute the convolution of the two corresponding vectors $\vec{a}$ and $\vec{b}$ in the compressed transform domain $\mathcal{T}_1$. Let $p_a(x) = \sum_{i=0}^{n-1} a_i x^i$ be the corresponding sparse polynomial of $\vec{a}$ and $p_b(x) = \sum_{i=0}^{n-1} b_i x^i$ the sparse polynomial of $\vec{b}$. We evaluate $p_a(x)$ at $2n'$ evenly spaced roots of unity. Specifically, let $\omega_{2n'}$ be the $2n'$-th root of unity We will evaluate $p_a(x)$ at $2n'$ points: $\omega_{2n'}^{-i}$ where $0 \leq i \leq 2n' - 1$. We interpolate a sparse polynomial $h(x)$ of $2n'$ non-zero terms and degree $2n$ such that $h(x_i) = p_a(x_i)p_b(x_i)$ for each $x_i = \omega_{2n'}^{-i}$, where $0 \leq i \leq 2n' - 1$. By this sparse polynomial interpolation and the convolution theorem, the coefficients of h can be shown to be the convolution of the two vectors $\vec{a}$ and $\vec{b}$ in the compressed transform domain $\mathcal{T}_1$. To insure the required compression, we keep all coefficients of h of degree $< k$ and keep only s of the coefficients of h of largest magnitude of degree $\geq k$. We use this resulting vector $\hat{h}$ to approximate the DFT of the dot product of the two vectors $\vec{u}$ and $\vec{v}$, as required. Thus, by use of known sparse polynomial evaluation and interpolation algorithms (see [2, 3, 10]), the total cost of computing the dot product is $O(n' \log n' + \log n)$. We denote the approximated computation of dot product in the transform domain by $ApproxDotProd()$. We summarize the above results in the following theorem:

Theorem 1 *The following operations on vectors in the untransformed domain $\mathcal{U}_1$ can be approximately performed in the compressed transform domain $\mathcal{T}_1$: (i) the operations of addition and convolution, in $O(n')$ time and (ii) the dot product, in $O(n' \log n' + \log n)$ time, where n is the uncompressed size (the length of the vector in the untransformed domain $\mathcal{U}_1$) and n' is the compressed size (the number of non-zero coefficients of the vectors in the compressed transform domain $\mathcal{T}_1$).*
Note that the error and degradation due to lossy compression depends on the degree of compression factor just as in standard lossy compression algorithms. The above computations may not be appropriate for general algebraic computations if vectors can not be lossily compressed with small error. However, in many 1D applications such as speech processing the input data is known for a large class of inputs to have excellent lossy compressibility, with small error, using similar compression techniques. So for these applications, our techniques for computing in the compressed transform domain can be applied to produce a reduction in the amount of computation for the solution of problem with limited error. We next extend the above method for computation in the compressed transform domain to 2D.

2.1 Extension to Two-dimensions

The operations in compressed transform domain are readily extendible to two dimension. We will summarize this extension in the following theorem:

Theorem 2 *The following operations on arrays of size $n \times n$ in the untransformed domain $\mathcal{U}_2$ can be approximately performed in the compressed transform domain $\mathcal{T}_2$: (i) the operations of addition and convolution in cost $O(n')$ time and (ii) the dot product with cost $O(n' \log n' + n \log n)$ time, where the arrays are of size $n \times n$ in the 2 dimensional untransformed domain $\mathcal{U}_2$ is and n' is the number of non-zero coefficients of the arrays in the 2 dimensional compressed transform domain $\mathcal{T}_2$.*

Note that (as in the 1D case), the error and degradation due to lossy compression depends on the degree of compression factor just as in standard lossy compression algorithms. In our applications of choice, such as image processing in $2D$, the input data is known for a large class of inputs to have excellent lossy compressibility, with small error, using similar compression techniques. In this paper, we apply this method to the volume rendering problem where the resulting image is already known to be highly compressible, and so we can greatly reduce the amount of computation by avoiding costly operations in the original domain by computing directly using the compressed forms of the images in the transform domain. We apply this generalized 2D method to the volume rendering problem where the resulting image is already known to be highly compressible. So we can greatly reduce the amount of computation by avoiding costly operations in the original domain by computing directly using the compressed forms of the images in the transform domain. The 2D DFT_n of an array in the original domain can be obtained by computing one-dimensional DFT by rows and then computing one-dimensional DFT for the resulting array's columns [14]. Let $\omega_{2n} = \exp(\pi\sqrt{-1}/n)$ be the $2n$th root of unity over the complex numbers. Let M be an $n \times n$ array such that $M = (m_{i,j})$. Consider the bivariate polynomial $Q(x, y) = \sum_{i,j=0}^{n-1} m_{ij} x^i y^j$ with variables x and y. Then the 2D DFT_{2n} of M is the $2n \times 2n$ array A where $A_{i,j} = Q(\omega_{2n}^i, \omega_{2n}^j)$ $= \sum_{i,j=0}^{n-1} m_{i,j}\omega_{2n}^{i+j}$, that is $A_{i,j}$ is the evaluation of $Q(x, y)$ at $x = \omega_{2n}^i$ and $y = \omega_{2n}^j$. Using Fast Fourier Transform the DFT and inverse DFT of an $n \times n$ array can be computed in $O(n^2 \log n)$ time. We compress the transformed array A using a method similar to the one-dimensional case. We assume, w.o.l.g., that k is a perfect square and keep a dense $\sqrt{k} \times \sqrt{k}$ subarray of elements of the transformed array (those terms that are in both first $\sqrt{k}$ rows and first $\sqrt{k}$ columns). These coefficients correspond to the low frequency terms in the original image. We keep s remaining entries in the array of highest magnitude. Let $P(x, y) = \sum_{i,j=0}^{n-1} a_{ij} x^i y^j$ be the corresponding sparse bivariate polynomial of variables x, y. Let $n' = s + k$. We assume w.o.l.g. $2n'$ is a perfect square. The total storage space needed is ρn^2 where $\rho = \frac{n'+s}{n^2}$, since we need to store the position indices of s terms of the highest magnitude as well. (Our compression scheme is similar to the JPEG image compression scheme in that only the high frequency (sparse) terms which contains non-vital information are compressed. The similarity is advantageous, since it indicates that we may obtain compression ratio comparable to to that of conventional methods such as JPEG. The main difference between our compression scheme and JPEG compression is that JPEG and other related image transform compression schemes is that they apply the transform only on blocks of consecutive subsubarrays of some bounded size. In practice, to further decrease our computational costs, we can also apply this decomposition method, which limits computational costs.) A *sparse bivariate polynomial* is a polynomial of degree n in each of two variables, say x, y which has s non-zero terms where $s \ll n$. The compressed transform array A in the transform domain has a corresponding sparse polynomial $p(x) = \sum_{i=0}^{n-1} a_i x^i$ whose coefficients form the same array. It is known (see [2, 3, 10]) that the results for evaluation and interpolation of a sparse univariate polynomial with degree n and s terms at s arbitrary points, hold also for bivariate polynomials, so again evaluation and interpolation can be done in $O(s \log^2 s + n \log n)$ time. Furthermore, the cost decreases to $O(s \log s + n \log n)$ if the evaluation or interpolation points are the s roots of unity or s chirp points. We consider the problem how to execute operations such as addition, convolution and dot product on arrays in the original domain $\mathcal{U}_2$ by using their corresponding representations in a two dimensional compressed transform domain $\mathcal{T}_2$. Without loss of generality, we consider the operations on two $n \times n$ arrays M, M' using their corresponding arrays A, A' in the 2D compressed transform domain $\mathcal{T}_2$, which are encoded using the compression method described above. Let $n' = s + k$ be the total number of coefficients stored by A, and A', where s is number of sparse high order terms stored and the dense subarray of low frequency terms stored is of size $\sqrt{k} \times \sqrt{k}$. We define the *dot product* of two arrays M, M' to be the array M'' such that $M''_{i,j} = M_{i,j} M'_{i,j}$ (note that this definition differs from some other definitions, but is convenient for our applications). The result is to be stored in the 2D compressed transform domain $\mathcal{T}_2$. Again, the approximate addition and convolution of two $n \times n$ compressed arrays can be performed easily in $O(n')$ time in the compressed transform domain $\mathcal{T}$, just as previously described for one-dimensional vectors. By the *2D Convolution Theorem*, the convolution of two $n \times n$ arrays M, M' can be computed in the transform domain using the 2D DFT_{2n}, performing dot product on the transformed DFT_{2n} arrays A, A' and then computing the inverse 2D Discrete Fourier Transform (DFT_{2n}^{-1}) of the resulting array, both taking $O(n^2 \log n)$ time. This is routine for image processing applications. In our applications, we need not apply the forward or inverse 2D Discrete Fourier Transform, and instead simply perform dot product on the prior transformed DFT_{2n} arrays A, A' in time $O(n')$ proportional to the number of stored coefficients n'. We again denote the approximated computation of addition and convolution in the transform domain by *ApproxAdd*() and *ApproxConv*(). Again, the convolution operation and dot product operation are dual problems in original and transform domain. The dot product of two $n \times n$ arrays M, M' convolution of two arrays A, A' given in the transform domain $\mathcal{T}_2$ can be computed by computing an inverse 2 D DFT of the two arrays, computing a dot product of the resulting arrays A, A' in the original domain $\mathcal{U}_2$, and then converting the result back into the 2D DFT$_{2n}$ in $\mathcal{T}_2$. Again, the conversion to and from these domains $\mathcal{U}_2, \mathcal{T}_2$ appears to be required if we wish to do combinations of dot product and convolution operations, but we again avoid this by use of sparse polynomial evaluation and interpolation. Let $p_A(x, y) = \sum_{i,j=0}^{n-1} A_{i,j} x^i y^j$ be the corresponding sparse polynomial of array A and $p_{A'}(x, y) = \sum_{i,j=0}^{n-1} A'_{i,j} x^i y^j$ the sparse polynomial of array A'. Let $\omega_{2n'}$ be the $2n'$-th root of unity. We evaluate $p_A(x, y)$ at $2n'$ evaluation points; that is at all $x_i = \omega_{2n'}^i$ and $y_j = \omega_{2n'}^j$ for each $0 \le i, j \le \sqrt{2n'} - 1$. We interpolate a sparse bivariate polynomial $h(x, y)$ of $2n'$ non-zero terms and degree $\le 2n'$ in each of the variables x, y such that $h(x_i, y_j) = p_a(x_i, y_j) p_b(x_i, y_j)$ for each $x_i = \omega_{2n'}^{-i}$, and each $y_j = \omega_{2n'}^{-j}$ where $0 \le i, j \le \sqrt{2n'} - 1$. By this sparse polynomial interpolation and the convolution theorem, the coefficients of h can be shown to be the convolution of the two arrays A, A' in the compressed transform domain $\mathcal{T}_2$. To insure the required compression, we keep all coefficients of h of degree $< \sqrt{k}$ in each of x, y and keep only s of the coefficients of h of largest magnitude of degree $\ge \sqrt{k}$ in each of x, y. We use this resulting vector $\hat{h}$ to approximate the DFT of the dot product of the two arrays M and M', as required. Thus, by use of known sparse bivariate polynomial evaluation and interpolation algorithms (see [2, 3, 10]), the total cost of computing the dot product of the two $n \times n$ arrays is $O(n' \log n' + n \log n)$, which is $O(\rho n^2 \log n)$, where $\rho = \frac{n'+s}{n^2}$, assuming $\rho \ge 1/n$.

3 Volume Rendering Algorithms

In recent years, scientific visualization has become a fast growing area in image processing which is concerned with various techniques to help scientists and engineers to extract meaningful information from large set of data from simulations and experimentations. The volume rendering problem is to directly display image of scalar and vector data samples defined in three or higher dimensions. It is one of the most actively researched subfields of scientific visualization. Some of the most commonly used direct volume rendering algorithms are ray casting [12, 15], splatting [17] and volume shearing [8, 11]. Among these algorithms, the splatting algorithm is the most efficient. However, due to the large size of the input data set, it is important to further decrease the amount of computation of splatting algorithms.

3.1 Feed-backward and Forward mapping algorithms

There are two primary approaches to solving the volume rendering problem: (i) the feed-backward methods that map the image space onto the data set, and (ii) the feed-forward methods that map each volume element onto the image plane. The feed-forward methods are more favorable than the feed-backward methods because in feed-forward methods each data sample only needs to know about a small surrounding neighborhood of other samples, not the whole data set as in the feed-backward methods. This makes the feed-forward methods readily to be implemented in parallel environment.

Forward mapping algorithms using the feed-forward methods normally consist of four steps: reconstruction, transformation, shading and resampling. The image renderer reconstructs the continuous volume function from the input samples by convolving the samples with a carefully chosen reconstruction filter kernel. The volume function is then transformed into image space and the transformed data is shaded to reflect individual sample's contribution to the final image. Finally the sampling process generates a regular collection of discrete data values from a continuous signal.

The processes of sampling and reconstruction are central to the volume rendering process because the deviation from the ideal case in both processes causes errors in the final results. Furthermore, most of the computation occurs in the reconstruction process. Therefore it is desirable to design a volume rendering algorithm that speeds up the reconstruction process by reducing the computation time while minimizing the amount of errors caused by different sources.

3.2 The Splatting Algorithm for Volume Rendering

We give here a brief description of the volume rendering problem and the well-known splatting algorithm for solving volume rendering problem for those who are not familiar with the problem or with the splatting algorithm (also see [8, 13, 17, 18]). Westover [17] presented a feed-forward volume rendering algorithm named "splatting". The name is derived from the description of the algorithm where each individual sample contributes to the overall image one-by-one, a process similar to throwing snowballs at a thick wall making "splat" sounds. In the actual algorithm, the term "splat" refers to the process of determining a sample's image-space footprint on the image plane, and adding the sample's effect over that footprint to the image. In the splatting algorithm, the sample is treated as a reflective, light-emitting, semi-transparent cloud. The sample is first *classified* to determine the discrete values for the primary properties that represent the sample. Then it is *illuminated* using certain illumination model. The process of classifying a sample and applying an illumination model to calculate the sample's illumination effects is called the *CRIO* process. The splatting algorithm consists of four main parts: transforming, CRIO, reconstruction and visibility. The transformation process converts the input tuple's mesh space $< i, j, k >$ into an image space $< x, y, z >$. The renderer then runs the CRIO process on the image-space tuple to generate a CRIO tuple for each data sample which contains information of color, opacity and image coordinates for this sample. Only the data sample whose opacity is not zero is passed to the reconstruction process. The reconstruction process determines the image-space contribution of each CRIO tuple to produce a single splat tuple. Finally the renderer combines the splat tuples to form the image using certain visibility rule. The visibility rules are different for front-to-back and back-to-front traversals, but the rules are equivalent. Formally, let I_{xy}^{ℓ} denote the output intensity of image at point $< x, y >$ after ℓ-th iteration, let F_{xy}^{ℓ} denote the intensity of data sample at point $< x, y, \ell >$ calculated in the reconstruction process, let A_{xy}^{ℓ} denote the output opacity of image at point $< x, y >$ after ℓ-th iteration, and let D_{xy}^{ℓ} denote the opacity of data sample at point $< x, y, \ell >$ calculated in the reconstruction process using the footprint function. The algorithm accumulates the intensity I_{xy} at image point $< x, y >$ by adding contribution F_{xy}^{ℓ} of each data sample at coordinates $< x, y, \ell >$. For example, the intensity that a data sample at coordinates $< x, y, \ell >$ contributes is computed by the amount of light emitting from the data sample $F_{xy}^{\ell} \times D_{xy}^{\ell}$ times a factor of $1 - A_{xy}^{\ell-1}$ (the portion that is not blocked by current opacity of image point $< x, y >$). The opacity at image point $< x, y >$ is increased by the amount of light that goes through by the previous opacity $1 - A_{xy}^{\ell-1}$ but is blocked because of the opacity of the new data sample D_{xy}^{ℓ}. The formulae for the back-to-front traversal can be explained similarly. For a front-to-back traversal, the algorithm first initialize the intensity and opacity arrays of image and then compute the output intensity and opacity incrementally, as follows:

```
    **Splatting Volume Rendering:**
for $x = 1...n$; for $y = 1...n$ do $I^0_{xy} = 0, A^0_{xy} = 0$
for $\ell = 1...n$ do for $x = 1...n$; for $y = 1...n$ do
    $I^\ell_{xy} = I^{\ell-1}_{xy} + ((1 - A^{\ell-1}_{xy}) \times (F^\ell_{xy} \times D^\ell_{xy})), A^\ell_{xy} = A^{\ell-1}_{xy} + ((1 - A^{\ell-1}_{xy}) \times D^\ell_{xy})$
    Output $I^n, A^n$ .
```

The algorithm works similarly, for a back-to-front traversal, as follows:

```
for $x = 1...n$; for $y = 1...n$ do
            $I^{n+1}_{xy} = 0, A^{n+1}_{xy} = 0$
for $\ell = n, n-1...1$ do
        for $x = 1...n$; for $y = 1...n$ compute
            $I^\ell_{xy} = ((1 - A^{\ell+1}_{xy}) \times I^{\ell+1}_{xy}) + (F^\ell_{xy} \times D^\ell_{xy})$
            $A^\ell_{xy} = ((1 - A^{\ell+1}_{xy}) \times D^\ell_{xy}) \times A^{\ell+1}_{xy})$
    Output $I^1, A^1$.
```

3.3 Footprints

Among these processes, the reconstruction part where the renderer must determine each sample's contribution to the final image is most compute-intensive. The reconstruction is normally performed by convoluting the sample function with the kernel function. The kernel function is a discrete array is carefully chosen to balance the tradeoff between the amount of computation and the image quality. For example, a high-quality and computationally expensive kernel is the one using the Gaussian density function ϕ which is defined by

$$\phi(x) = e^{-\frac{x^2}{2\sigma^2}}. \tag{2}$$

Instead, in the reconstruction process of the splatting algorithm, a *footprint table* is built for each data sample to spread the sample's energy onto the image plane. Since the footprint function is constant for all input samples, the renderer builds a footprint table at the beginning of the reconstruction process and then uses it for every sample. One easy way to build the footprint table is to determine the image-space extend of the projection of the reconstruction kernel and then select a sub-pixel sampling rate. However, the renderer must still integrate the reconstruction kernel to build the footprint table once per image. An alternative way is to model the result of file=splat2.ps,angle=-90

Figure 1: Reconstruction process using compressed FFT.

the integration with some simple function, for example, a Gaussian, since the result of integrating one dimension of a three-dimensional Gaussian is still a Gaussian. This method evaluates the simple function at the table entries without integration. The kernel needs to be truncated in this case since the extent of the kernel is infinite. Formally, let $f()$ be the input samples, $h()$ be the reconstruction kernel, and $g()$ be the reconstructed signal, the volume reconstruction equation for a discrete input, $f(i, j, k)$, is

$$g(x, y, z) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} f(i, j, k)h(x - i, y - j, z - k). \tag{3}$$

The footprint function centered at the origin of three dimensional space is defined as

$$f(x, y) = \int_{-\infty}^{\infty} h(x, y, z)dz. \tag{4}$$

Thus the footprint function defined as above can be viewed as a two-dimensional image-space projection of the reconstruction kernel. The footprint table is built only once at the beginning of each rendering process and the renderer uses the same footprint table for every sample. After the contribution of each sample to the image has been determined, for each pixel in the footprint, the renderer weights the sample by the footprint value and composite the sample's color and opacity into the *accumulation buffer*. The compositing can be accomplished by either front-to-back or back-to-front traversal and a *sheet buffer* is used to correctly calculating the value of a reconstructed point that lies between samples with overlapping kernels. After the image is constructed, the user has the freedom to interactively change viewing parameters and the renderer regenerates image by recomputing the footprint table.

4 Volume Rendering in Compressed Fourier Transform Domain

We apply our computation methods described above to volume rendering problem. We present our application of computation in compressed transform domain to the splatting algorithm in this section. In our volume rendering algorithm, the original image data sample and the footprint are all stored in the compressed transform domain $\mathcal{T}_2$. The outputs will also be stored in the compressed transform domain $\mathcal{T}_2$. In the reconstruction process of our algorithm, we consider the input sample data as a 3-D $N \times N \times N$ array σ_{xy}^{ℓ} which is stored in compressed transform domain $\mathcal{T}_2$. Similarly, the footprint function f_{xy} defined in the previous section is stored in the compressed transform domain $\mathcal{T}_2$. The approximate convolution $\hat{F}$ of the data sample and the footprint in the spatial domain can be performed as dot product in the transform domain $\mathcal{T}_2$, and stored in this domain. Specifically, $\hat{F}$ can be computed using operation $ApproxConv(\sigma, f)$ in the compressed transform domain $\mathcal{T}_2$ as described in Section 2, where σ is the image data sample, and f is the footprint function. Let $\rho \geq 1/N$ be the compression factor. Then total computation time for this approximate convolution is $O(\rho N^3)$. The key operations in the image rendering process are addition and dot product of two-dimensional arrays in the uncompressed domain $\mathcal{U}_2$. Therefore, the image rendering in the splatting algorithm can also be performed in compressed transform domain $\mathcal{T}_2$ using the 2-D array operations described in section 2. In our algorithm, we assume that the inputs are the intensity arrays and opacity arrays stored in compressed transform domain $\mathcal{T}_2$. We denote the intensity arrays by $\hat{F}$ which is a compressed representation in transform domain $\mathcal{T}_2$ of the intensity arrays F in the untransformed domain $\mathcal{U}_2$. Similarly, we denote the opacity arrays by $\hat{D}$ which is a compressed representation in $\mathcal{T}_2$ of the intensity arrays D in $\mathcal{U}_2$. We let $\hat{I}^{\ell}$ be the output image intensity at the ℓ-th level which is in compressed transform domain $\mathcal{T}_2$. Also we let $\hat{A}^{\ell}$ be the output opacity of image at the ℓ-th level which is in compressed transform domain $\mathcal{T}_2$. We compute the operations (addition, dot product and convolution) using 2D approximated computations described in Section 2. We assume that the input and output are both stored in compressed transform domain $\mathcal{T}_2$ and we analyze the computational cost of our algorithm accordingly. Applying a front-to-back traversal similar to the splatting algorithm, our rendering algorithm for front-to-back traversal in the compressed transform domain is the following:

```
**Volume Rendering in Compressed Transform Domain**
$\hat{I}^0 = 0, \hat{A}^0 = 0$; **for** $\ell = 1...n$; **do**
    $\hat{I}^{\ell} = ApproxAdd(\hat{I}^{\ell-1}, ApproxDotProd(1-\hat{A}^{\ell-1}, ApproxDotProd(\hat{F}^{\ell}, \hat{D}^{\ell})))$
    $\hat{A}^{\ell} = ApproxAdd(\hat{A}^{\ell-1}, ApproxDotProd(1-\hat{A}^{\ell-1}, \hat{D}^{\ell}))$
    **Output** $\hat{I}^n, \hat{A}^n$.
```

Also, applying a back-to-front traversal our rendering algorithm for back-to-front traversal in the compressed transform domain is as follows:

```
$\hat{I}^{n+1} = 0, \hat{A}^{n+1} = 0$; **for** $\ell = n, n-1...1$; **do**
    $\hat{I}^{\ell} = ApproxAdd(ApproxDotProd(1-\hat{A}^{\ell+1}, \hat{I}^{\ell+1}), ApproxDotProd(\hat{F}^{\ell}, \hat{D}^{\ell}))$
    $\hat{A}^{\ell} = ApproxDotProd(ApproxDotProd(1-\hat{A}^{\ell+1}, \hat{D}^{\ell}), \hat{A}^{\ell+1})$
    **Output** $\hat{I}^1, \hat{A}^1$.
```

4.1 Performance Analysis.

We consider the performance of our algorithm for volume rendering on a given $N \times N \times N$ three-dimensional data array. In the original splatting algorithm, the dominant part of the time complexity is the reconstruction phase. Let f be the size of the footprint (normally say 15×15 pixels) and P_I (a constant say $1/2$) be the frequency of the non-zero intensity (non-transparent) volume pixels which contributes to the final image.

Since the integration is performed along all three dimensions, the convolution of the sample function and the kernel takes time $O(N^3 f P_I)$. Because P_I is a constant, the time complexity is $O(N^3 f)$. The size of the footprint f is the major factor in the tradeoff between the computation time and the resulting image quality. To demand a higher image quality, the corresponding larger size of the footprint may cause the computation to be too expensive. In the preprocessing stage of our algorithm, the $N \times N \times N$ spatial data array and the footprint function are all transformed into transform domain using Fast Fourier Transform in 3-D which takes $O(N^3 \log N)$ time and compressed using the method described in Section 2. This preprocessing needs to be done only once.

By computing in compressed transform domain, the computation of the reconstruction (convolution) stage can be reduced to $O(\rho N^3)$ where ρ is the compression factor. In the final rendering process, the time complexity is $O(\rho N^3 \log N)$ where ρ is the compression factor. Therefore, the time complexity for our algorithm is $O(\rho N^3 \log N)$, given this preprocessing. The volume rendering algorithm using compressed transform domain we presented has several advantages over the original splatting algorithm. Because our algorithm works in compressed transform domain instead of the original spatial domain, the reconstruction process is easier to visualize. The convolution in the spatial domain corresponds to a dot product in the frequency domain. By transforming the sample data into Fourier transform, we avoid the computational cost of convolution.

Our volume rendering algorithm takes $O(\rho N^3 \log N)$ time while the original splatting algorithm runs in $O(N^3 f)$ where f is the size (number of non-zero terms) of footprint. In practice, the compression factor ρ is typically 1/30 because image data can be lossily compressed well. Since the typical image size is 1000×1000 pixels, $\log N$ is approximately 10. The footprint size is normally around 200 [17].

There are constant factors in the complexities of both the splatting algorithm and our algorithm but these constant factors are similar, say about 5 to 10. Therefore, our algorithm may improve the running time of splatting algorithm by a significant factor in practical use. Furthermore, the preprocessing is only done once for any given input data set. The transformed sample data is used for subsequent repeated image construction. Since each individual image rendering is very fast, it allows the user to change parameters of the rendering algorithm and view the result after a short time. Therefore, the progressive refinement of the final image is a natural consequence of the transform method.

By using the compressed transform representation, the renderer can build preview images generated by computing a highly compressed transform representation of the sample data in the frequency domain. The computation time decreases as the image data becomes more compressed. When a researcher uses the renderer in an interactive way and wants to get feedbacks whenever any viewing parameters change, the renderer can provide an image, which is not necessary of high quality but good enough to reflect the parameter changes, by choosing a proper compression factor for the compressed transform representation. This preview update in our algorithm is much faster than the original splatting algorithm where the convolution has to be computed expensively whenever any view parameters change.

5 Conclusion

We proposed a new computation method using compressed representation of inputs in transform domain. We applied this method to the splatting volume rendering problem and designed an improved algorithm which saves computation time by performing operations in compressed transform domain. This new method can also provide similar speed up when view parameters are changed. In practical implementation, we suggest that the DCT or FHT is used instead of FFT to further reduce storage requirement and improve running time. We expect that the same methodology of computing in compressed transform domain can be applied to speed up other algorithms in areas such as image and speech processing.

References

[1] A.Amir and G.Benson and M. Farach. Let sleeping files lie: pattern matching in Z-compressed files. *Symposium on Discrete Algorithms*, 1994.

[2] D. Bino and V. Pan. *Polynomial and matrix computations*, V 1, Birkhauser.

[3] A. Borodin and P. Tiwari. On the Decidability of Sparse Univariate Polynomial Interpolation, *Proc. 22nd Ann. ACM Symp. on Theory of Computing*, 535–545, ACM Press, New York, 1990.

[4] P.Cignoni and L.D.Floriani and C.Montani and E.Puppo and R.Scopigno, Multiresolution modeling and visualization of volume data based on simplicial complexes, *Proceedings of Symposium on Volume Visualization*, 1994.

[5] R.J. Clarke, *Transform coding of images*, Academic Press, 1985.

[6] T.H.Cormen, C.E.Leiserson and R.L.Rivest, *Introduction to algorithms,* McGraw-Hill Book Company, 1990.

[7] S. Chen and J. H. Reif. Using difficulty of prediction to decrease computation: Fast sort, priority queue and computational geometry for bounded-entropy inputs. *34th Symposium on Foundations of Comp. Sci.*, 104–112, 1993.

[8] J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. *Proceedings of the 1992 Workshop on Volume Visualization*, pages 91–106, Boston, 1992.

[9] D. Dudgeon and R.Mersereau. *Multidimensional Signal Processing.* Prentice-Hall, N.J., 1984, pp.81, 82, 363–383.

[10] E. Karatsuba and Y.N. Lakshman and J.M. Wiley, Modular Rational Sparse Multivariate Polynomial Interpolation, *Proc. ACM-SIGSAM Int. Symp. on Symb. and Alg. Comp. (ISSAC '90)*, 135–139, ACM Press, New York, 1990.

[11] P. Lacroute and M.Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Comp. Graphics*, 28(4):451–458, 1994.

[12] M. Levoy. Display of surface from volume data, *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[13] T. Malzbender. Fourier volume rendering. *ACM Transactions on Graphics*, 12(3):233–250, July 1993.

[14] W.B. Pennebaker and J.L. Mitchell. *JPEG still image data compression standard.* Van Nostrand Reinhold, 1992.

[15] P. Sabella. A rendering algorithm for 3d scalar fields. *Computer Graphics,* 22(4):51–58, 1988.

[16] C. Upson and M. Keeler. V-buffers: Visible volume rendering. *Computer Graphics,* 22(4):59–64, 1988.

[17] L.A.Westover. Splatting: a parallel, feed-forward volume rendering algorithm, Ph.D thesis, University of North Carolina, Dept. of Comp. Sci., 1991.

[18] L.Westover, Footprint evaluation for volume rendering, *Computer Graphics,*, Volume 24, Number 4, August 1990.

[19] J.Wilhelms and A.V.Gelder. Muti-dimensional trees for controlled volume rendering and compression, *IEEE Proc. Symp. on volume visualization*,1994.