# Shear-Warp Deluxe: The Shear-Warp Algorithm Revisited

Jon Sweeney and Klaus Mueller

Department of Computer Science, State University of New York at Stony Brook

**Abstract**

*Despite continued advances in volume rendering technology, the Shear-Warp algorithm, although conceived as early as 1994, still remains the world's fastest purely software-based volume rendering algorithm. The impressive speed of near double-digit framerates for moderately sized datasets, however, does come at the price of reduced image quality and memory consumption. In this paper, we present the implementation and impact of certain measures that seek to address these shortcomings. Specifically, we investigate the effects of: (i) post-interpolated classification and shading, (ii) matched volume sampling on zoom, (iii) the interpolation of intermediate slices to reduce inter-slice aliasing, and (iv) the re-use of encoded RLE runs for more than one major viewing direction to preserve memory. We also study a new variation of the shear-warp algorithm that operates on body-centered cubic grids. We find that the reduction of the number of voxels that this grid affords translates into direct savings in rendering times, with minimal degradation in image quality.*

## 1. Introduction

Volume graphics is a graphics technology that has gained immense momentum in recent years. In volume graphics the object exists discretized on a 3D raster, which is in stark contrast to polygonal graphics where the object is represented as a hull of polygons. A steadily growing list of applications has emerged that show great benefits from maintaining a volumetric data representation. A few of these are listed here: medical diagnosis and surgical simulation, CAD/CAM prototyping and industrial design, oil and gas exploration, virtual sculpting, teaching of biology and chemistry in high schools, computer games and special effects for movies, scientific data exploration, and information visualization, such as for business, stock market, and economy.

However, up to this date, it has remained a great challenge to perform high-quality volume rendering at interactive framerates. But interactive display is an absolute necessity if volume graphics is to become a mainstream graphics technology. Users have simply become too spoiled by the amazing speed of polygonal graphics to accept renderers that don't produce at least ten frames/s for these type of applications. There are four main paradigms in which volume rendering is performed in current days: raycasting [12][27], splatting [28], shear-warp [10], cell-projection [15][25], texture-mapping hardware-assisted [4][7][21], and via custom hardware [16][20]. At this time, only custom hardware can achieve interactive framerates in excess of 30 frames/s while still providing excellent image quality. The approaches that use modern PC graphics cards, such as the GeForce3 and the ATI Radeon have come closer in quality in recent years and have also demonstrated high framerates, but at a much reduced cost. Raycasting and splatting, on the other hand, have shown images of superior quality, but are usually not interactive [17]. A notable exception is the system devised by Knittel [8], which exploits Intel's MMX and Streaming SIMD in conjunction with aggressive caching to achieve interactive framerates. However, due to the limited graphics memory capacity of any of the hardware-based approaches and perhaps cost constraints, there still are certain advantages to employing a software-based renderer.

The shear-warp algorithm is a purely software-based renderer, although some of its concepts have found a hardware implementation in the VolumePro500 volume rendering board [20]. Shear-warp was invented by Lacroute and Levoy [10] and can be considered a hybrid between image-order algorithms, such as raycasting, and object-order algorithms, such as splatting. In shear-warp, the volume is rendered by a simultaneous traversal of RLE-encoded voxel and pixel runs, where opaque pixels and transparent voxels are efficiently skipped during these traversals. Further speed comes from the fact that a set of interpolation weights is pre-computed per volume slice and stays constant for all voxels in that slice. The caveat is that the image must first be rendered from a sheared volume onto a so-called base-plane, aligned with the volume slice most parallel to the true image plane (see Fig. 1). After completing the base-plane rendering, the base plane image is warped onto the true image plane and the resulting image is displayed. All of this combined enables framerates in excess of 10 frames/s on current PC processors, for a $128^3$ volume.

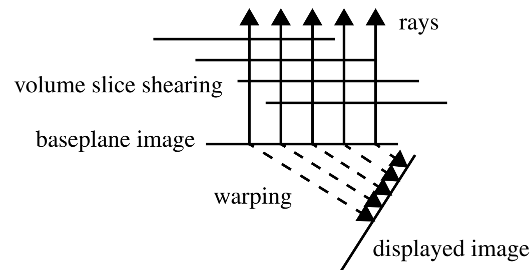



**Figure 1:** *A sketch of the shear-warp mechanism.*



delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

However, there are a number of compromises that had to be made in the process:

- Since the interpolation only occurs within one slice at a time, more accurate tri-linear interpolation reduces to less accurate bi-linear interpolation and the ray sampling distance varies between 1 and $\sqrt{3}$ , depending on the view orientation. This leads to aliasing and staircasing effects at viewing angles near 45˚.
- Since the volume is run-length encoded the authors use three sets of voxel encodings, one for the each major viewing direction. This triples the memory required for the runs, and it also causes visible switching artifacts when the major viewing direction changes. Similar popping artifacts can also be observed with the original implementation of splatting [28] when the major viewing axis along which compositing is performed changes at 45˚ (see [18] for more details).
- Since there is only one interpolated value per voxel-slice 4-neighborhood, zooming can only occur during the warping phase and not during the projection phase. This leads to considerable blurring artifacts at zoom factors greater than 2.
- Finally, the fact that voxels are stored and interpolated pre-shaded leads to additional blurring. This effect was demonstrated, for example, in [19].

All of these effects and shortcomings have already been demonstrated in the recent comparison paper by Meissner et.al. [17]. The work reported in the present paper seeks to devise efficient solutions to these shortcomings. Although it can be expected beforehand that these solutions will most probably lower rendering speed, it is unclear how much of a performance hit one will have to accept, and by how much rendering quality will improve. After all, we do not want to give up the hallmark-features of the shear-warp algorithm: The efficient in-slice interpolation scheme that affords the use of RLE-encoded runs, the pre-computed interpolation weights, and the post-rendering warp. Doing so will necessarily compromise image quality, so one should still not expect an image quality that rivals that of raycasting.

In our actual implementation, we also sought to employ “minimally invasive code surgery”, that is, we wanted to keep our changes simple, reusing as much of the original code as possible. We first re-implemented the original shear-warp algorithm at a fairly optimized level, but without some of the generality of Lacroute’s original implementation, volpack [1]. Then we added the new features, including the ability to render body-centered cubic grids. Body-centered cubic grids [6], or BCC grids for short, were recently discovered for their application in volume rendering by Theussl, Möller, and Gröller [26]. By storing the voxels in this grid arrangement (eight voxels at the vertices of a cube and one in its center) one can reduce the number of voxels by about 30%, but still maintain the same signal content in frequency space. We shall see more on the implementation of this grid in section 4.

Previous work related to the shear-warp algorithm includes two parallel implementations, one on a shared-memory machine [11] and one on a distributed-memory machine [2]. Also noteworthy is the 4D shear-warp extension by [3] and the recent implementation of an improved perspective shear-warp method in a virtual reality setting [24]. Finally, there are two prior works that have employed a shear-warp factorization, but without the RLE volume encoding [5][23].

Our paper is organized as follows: First, in section 2, we describe the shear-warp algorithm in closer detail, in the context of our modifications. Then, in section 3, we describe the new features and their implementation, while section 4 discusses the extension of the shear-warp algorithm to BCC grids. Finally, section 5 shows the results we were able to obtain, and section 6 concludes the paper.

## 2. Preliminaries

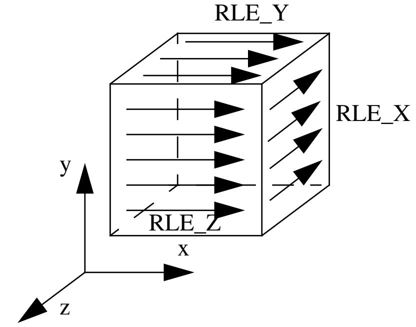
Consider Fig. 2 where we have attempted to construct a pseudo-code outline of shear-warp’s essentials. We have omitted a few details here for brevity and conciseness, and the interested reader may refer to Lacroute’s dissertation [9] for more detail. We will be expanding on this pseudo-code in subsequent sections, when we add our modifications and extensions.

The function Construct_Shade_Cube() produces a cubic reflection map that can be indexed by a shade index which in turn can be calculated from an un-normalized gradient vector [22]. The function RLE_Encode() produces RLE encodings (see insert in Fig. 2) of the voxel data for all three major viewing directions, given the current opacity transfer function AlphaTF. Each RLE voxel carries the voxel density and the shading index that can be employed to index the shading table during rendering. Depending on the viewing direction, the appropriate RLE encoding is chosen, the view matrix $M_{view}$ is factorized into the shear matrix $M_{shear}$ and the warp matrix $M_{warp}$, the shear parameters are calculated based on $M_{shear}$, and the rendering into the base image is begun. For each slice *k*, the offset into the base plane image is given by the shear parameters, and the interpolation weights for this slice are computed based on this offsets.

When the slice is rendered into the base plane image, two adjacent voxels runs, bot_run and top_run, are always traversed concurrently and the pixel run that falls inbetween these runs is updated. The position of the pixels with respect to the run voxels is the same for all pixels. This is why the interpolation weights can be pre-computed, and this is also why the base-plane image must have the same resolution as the voxel slices, no matter what the zoom factor may be. Next, the pseudocode lists the mechanism that achieves the two-way skipping of transparent voxels and opaque pixels. Whenever a non-opaque pixel within a non-transparent run of voxels is identified, the shading index is used to retrieve the diffuse and specular color from the shading table and the pixel is composited with the corresponding baseplane_image pixel. Following, the RLE encoding of the baseplane image opacities is updated. Finally, after rendering all slices in this manner, the baseplane image is warped into the displayed image, using the warp matrix $M_{warp}$.

We will now describe the modifications we have made to this standard algorithm.

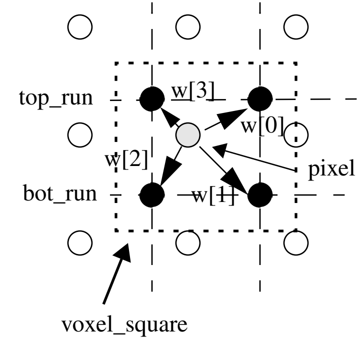
```
Shear_Warp (voxel_data)
    if New(M_view)
        shade_table ← Construct_Shade_Cube(M_view, L, E);
    if New(transfer_function)
        RLE_Encode(voxel_data, RLE_X, RLE_Y, RLE_Z);
    if (major_viewing_axis == X)  current_RLE ← RLE_X;
    if (major_viewing_axis == Y)   current_RLE ← RLE_Y;
    if (major_viewing_axis == Z)   current_RLE ← RLE_Z;
    Factorize(M_view, M_shear, M_warp);
    Calc_Shear_Parameters(M_shear, shear_u, shear_v, trans_u, trans_v);
    base_image ← Render(current_RLE);
    display_image ← Warp(base_image, M_warp);
```




```
Render (current_RLE)
    base_image.Initialize();
    num_scanline_pixels ← slice_width;
    num_scanlines ← slice_ height;
    for k ← front_slice ... end_slice, +1
        Composite_Slice (k);
    return(base_image);
```

```
RLE_Encode (voxel_data, RLE_X, RLE_Y, RLE_Z)
    for all voxels in voxel_data with AlphaTF[voxel.density] > 0
        RLE_voxel.shade_index ← Calc_Shade_Index  (voxel.gradient);
        RLE_voxel.density ← voxel.density;
        RLE_X.Add (RLE_X, RLE_voxel);
        RLE_Y.Add (RLE_voxel);
        RLE_Z.Add(RLE_voxel);
```

```
Composite_Slice (k)
    slice_u ← k · shear_u + translate_u;
    slice_v ← k · shear_v + translate_v;
    slice_u_int ← floor(slice_u);
    slice_v_int ← floor(slice_v);
    weights[4] ← Compute_Voxel_Weights(slice_u, slice_u_int, slice_v, slice_v_int);
    for j ← 0 ... num_scanlines-1, +1
        for i ← 0 ... num_scanline_pixels-1, +1
            bot_run ← Skip_Transparent_Voxels (i, j, k, current_RLE);
            top_run ← Skip_Transparent_Voxels (i, j+1, k, current_RLE);
            pixel_start ← Min (bot_run.start_voxel, top_run.start_voxel);
            pixel_end ← Max (bot_run.start_voxel + bot_run.length,
                                        top_run.start_voxel + top_run.length);
            for pixel ← pixel_start ... pixel_end, +1
                pixel ← Skip_Opaque_Pixels (pixel, j, base_plane_image.opacity_RLE)
                if (pixel > pixel_end)
                    break;
                voxel_square ← Get_Voxel_Square (bot_run, top_run, pixel);
                composited_opacity ← Composite_Pixel (voxel_square, weights);
                if composited_opacity > 0
                    Update_Opacity_RLE (pixel, base_plane_image.opacity_RLE);
            i ← pixel;
```




```
Composite_Pixel (voxel_square, weights)
    voxel_square.opacities ← AlphaTF[voxel_square.densities];
    pixel_opacity ← Interpolate (voxel_square.opacities, weights);
    if pixel_opacity > 0
        voxel_square.shades ← Get_Shades (shade_table, voxel_square.shade_indices);
        voxel_square.colors ← Calc_Colors (voxel_square.shades, voxel_square.densities, ColorTF);
        pixel_color ← Interpolate (voxel_square.colors, weights);
        composited_opacity ← Composite (pixel_color, pixel_opacity, base_plane_image.pixels);
    return (composited_opacity);
```

**Figure 2:** *Pseudocode of the standard shear-warp algorithm.*

## 3. The Modified Shear-Warp Algorithm

In this section we will elaborate on the four modification we have introduced into the standard shear-warp algorithm described in the previous section: (i) the re-use of encoded RLE runs for more than one major viewing direction to preserve memory, (ii) post-interpolated shading, (iii) matched volume sampling on zoom, and (iv) the interpolation of intermediate slices to reduce inter-slice aliasing.

### 3.1. Re-use of RLE runs for multiple major viewing axes

Consider the insert in Fig. 2 where we show the map of RLE encodings. From this drawing one can easily observe that the individual scanline runs of RLE_Z and RLE_Y are identical. The only difference is the order in which they are used, which is determined by the major viewing axis. If we keep an array of pointers into the RLE datastructure, one for each major viewing direction, then we can use RLE_Z for

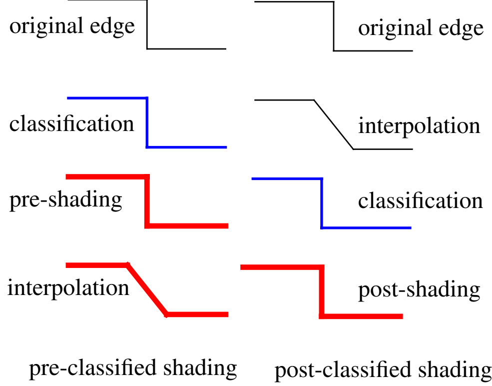



**Figure 3:** *An edge rendered both with pre-interpolated shading (left) and post-interpolated shading (right). The order of classification, shading, and interpolation is interchanged in these two variations. In post-interpolated shading, the blur left by the interpolation filter can be removed in the classification stage, while in pre-interpolated shading the interpolation is the final stage and the blur remains.*

both major viewing axes, y and z, and we neither have to compute nor store RLE_Y, however, we would still need RLE_X. Note that we could have instead combined RLE_Y and RLE_X, if we had ran RLE_Y along the z-axis. Since the scanlines accesses may be out of RLE sequence (for the y-axis in the above example), a map into the RLE datastructure is required to locate the RLE encoding of a given scanline. We use the array of pointers for this. Using such an array-indexed RLE encoding, the space savings are $O(n/3)$) where $n$ is the total number of non-zero voxels. Since the scanlines are sufficiently long and fill a cache line easily, a degradation in cache performance is unlikely.

### 3.2. Post-interpolated classification and shading

Traditional shear-warp uses pre-interpolated shading, i.e., all voxels are first classified and shaded, and the shaded voxels are then interpolated during rendering. However, interpolating a pre-shaded voxel neighborhood leads to blurred edges, especially on zooms. This was demonstrated, e.g., in [19] for splatting. The blurring will be less pronounced for shear-warp since a linear interpolation kernel of extent=2 grid spacings is used, in contrast to splatting's Gaussian kernel that has an extent of 4 grid spacings. Consider Fig. 3 for an illustration of this effect.

We can incorporate post-interpolation shading into the shear-warp algorithm relatively easy. In the current form each RLE voxel stores a density and a shading index, used to retrieve the diffuse and specular shading from the shading tables. For post-classification, we create a second table, called the normal table, of the same resolution than the shading table, where we store a cube map of normal vectors. Since both tables have the same resolution, we can still use the index stored in the RLE voxel, but we require one extra table lookup in the shading and compositing phase. The required modifications only affect the function Composite_Pixel(), and its new form Composite_Pixel_Post_Classify() is illustrated in Fig. 5. In terms of complexity the two versions are similar: Per pixel, the same number of interpolations are required. While post-classification requires the interpolation of a scalar density and a 3-vector normal, the traditional method interpolates an opacity scalar and a 3-vector color. Also, while the post-classified method only needs one sample to be colored, it requires the computation of an index into the shading table, based on the value of the interpolated gradient. This is about equivalent to coloring four samples in the standard method.

### 3.3. Matched volume sampling rate on zoom

The warp matrix $M_{warp}$ determines the stretching of the baseplane image. If the sampling rate of the displayed image is greater than that of the baseplane image then the frequency content of the baseplane image is not sufficient for the resolution of the displayed image and the displayed image will appear as if the rendered scene was lowpassed or blurred. This is illustrated in Fig. 5. There, $\Delta x$ and $\Delta y$, the grid spacings of the warped baseplane image, are both greater than the grid spacing of the displayed image. We can obtain $\Delta x$ and $\Delta y$ from the warp matrix. The warping equation is defined as:

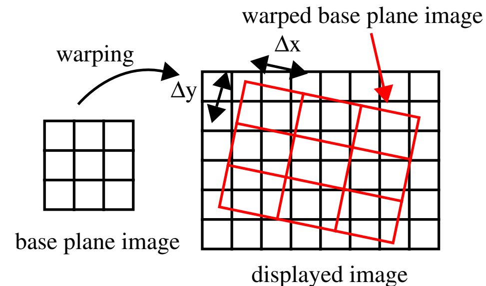



**Figure 5:** *The scaling of the warped base plane image.*

```
Composite_Pixel_Post_Classify (voxel_square, weights)
    pixel_density ← Interpolate (voxel_square.densities, weights);
    pixel_opacity ← AlphaTF[pixel_density];
    composited_opacity ← 0;
    if pixel_opacity > 0
        voxel_square.gradients ← Get_Normals (normal_table, voxel_square.shade_indices);
        pixel_gradient ← Interpolate (voxel_square.gradients, weights);
        pixel_shade_index ← Calc_Shade_Index (pixel_gradient);
        pixel_shade ← Get_Shade (shade_table, pixel_shade_index);
        pixel_color ← Calc_Color (pixel_shade, pixel_density, ColorTF);
        composited_opacity ← Composite (pixel_color, pixel_opacity, base_plane_image.pixels);
    return (composited_opacity);
```

**Figure 4:** *The modified routine Composite_Pixel_Post_Classify() that replaces Composite_Pixel() in the pseudocode of Fig. 2.*

$$\begin{bmatrix} x_{BI} \\ y_{BI} \end{bmatrix} = M_{warp} \begin{bmatrix} x_{DI} \\ y_{DI} \end{bmatrix} = \begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \end{bmatrix} \begin{bmatrix} x_{DI} \\ y_{DI} \end{bmatrix} \tag{1}$$

where $x_{BI}$ and $y_{BI}$ are the baseplane image coordinates, and $x_{DI}$ and $y_{DI}$ are the display image coordinates, respectively. From this we can compute Δx and Δy as follows:

$$\begin{aligned} \Delta x &= \sqrt{w_{00}^2 + w_{10}^2} \\ \Delta y &= \sqrt{w_{01}^2 + w_{11}^2} \end{aligned} \tag{2}$$

Thus, if we wanted to make the grid spacing of the warped baseplane in the displayed image about unity, then a baseplane sampling rate of $1/\Delta x$ and $1/\Delta y$ would be required during the rendering. A further constraint comes from the fact that only integer sampling rates place the sample points at constant positions within a square voxel neighborhood on a slice. The number of samples, $ns_x$ and $ns_y$, required per square voxel neighborhood is then written as follows:

$$\begin{aligned} ns_x &= \left\lceil \sqrt{w_{00}^2 + w_{10}^2} \right\rceil \\ ns_y &= \left\lceil \sqrt{w_{01}^2 + w_{11}^2} \right\rceil \end{aligned} \tag{3}$$

Thus, we require an array of $ns_x \cdot ns_y$ weights[4] arrays instead of the single weights[4] array in Fig. 2. Since we now have to render $ns_x \cdot ns_y$ sample points per voxel square, the rendering complexity is also increased by that amount. There may, however, be some savings due to the efficient runs traversals.

Finally, for the warp we need to account for the increased size of the baseplane image by dividing $w_{00}$ and $w_{10}$ by $ns_x$ and $w_{01}$ and $w_{11}$ by $ns_y$, respectively.

### 3.4. Intermediate slice interpolation

Due to the in-slice sampling of shear-warp, the inter-slice sampling rate varies depending on the viewing angle. Consider Fig. 6 where we show the 2D case. At 45˚ the distance between adjacent sampling points is $\sqrt{2}$, on views down a major diagonal it is $\sqrt{3}$. Basic rules of sampling theory tell us that we need at least a distance of 1.0 to combat aliasing artifacts and to remain within the Nyquist limit of the sampled signal. We can achieve this by interpolating one intermediate slice half-way between two adjacent volume slices. This is shown as the dotted line in Fig. 6.

We do not actually have to interpolate each intermediate slice, RLE-encode it and render it. We can do the inter-slice interpolation on the fly, using the existing RLE-encoded slices. For an illustration consider Fig. 7 where we show the 2D case. We can interpolate a sample point $s_{13,24}$ on intermediate slice $k+0.5$ as a two-step process from neighborhood voxels $v_1$, $v_2$, $v_3$, and $v_4$ on slice $k$ and $k+1$, respectively, given by the following expression:

$$\begin{aligned} s_{13,24} &= wt_{k+0.5,l} \cdot i_{13} + wt_{k+0.5,r} \cdot i_{24} \\ i_{13} &= 0.5v_1 + 0.5v_3 \\ i_{24} &= 0.5v_2 + 0.5v_4 \end{aligned} \tag{4}$$

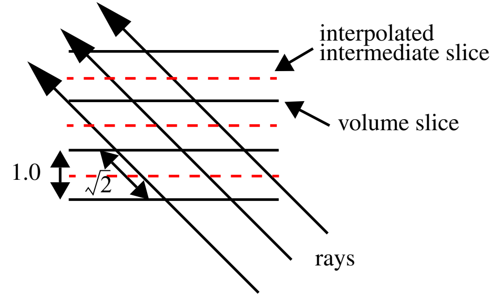



**Figure 6:** *Illustration of inter-slice undersampling and the interpolation of one intermediate slice per volume slice pair to resolve this. To show the actual sampling distance in volume space, we have left the volume slices unsheared.*

where $wt_{k+0.5,\,l}$ and $wt_{k+0.5,\,r}$ are the (left and right) weights that result from shearing the intermediate slice by an amount half way between that of slice $k$ and slice $k+1$. We can reorder this expression as follows:

$$\begin{aligned} s_{13,24} = {} & 0.5v_3 \cdot wt_{k+0.5,l} + 0.5v_4 \cdot wt_{k+0.5,r} \\ & + 0.5v_1 \cdot wt_{k+0.5,l} + 0.5v_2 \cdot wt_{k+0.5,r} \end{aligned} \tag{5}$$

Thus we can composite slice $k+0.5$ in a 2-step process: First we run through slice $k$ again, now weighted by a factor of 0.5 and with the interpolation weights set appropriate for slice $k+0.5$ (first line of (5)). Then we run through slice $k+1$, with identical interpolation weights and weighting (second line of (5)). Once we have added together the contributions of both slices, we can composite the result with the current baseplane image in the usual way.

This approach allows us to take advantage of the existing RLE-encoding of slice $k$ and $k+1$. Although we have not done so, one also could alter this algorithm slightly by simultaneously compositing slice $k$ and constructing the first line of (5), and do the same for slice $k+1$ and the second line of (5).

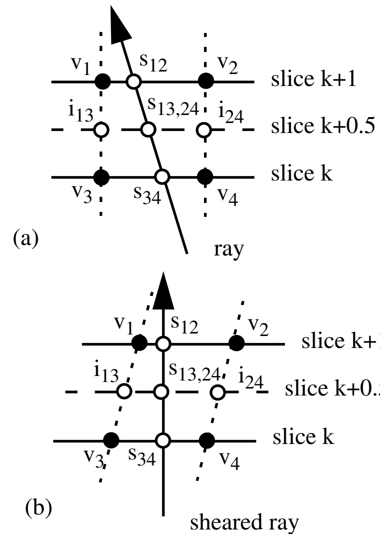



**Figure 7:** *Intermediate slice in (a) unsheared space and (b) sheared space.*

However, the time savings may be offset by the fact that run traversal would be sub-optimal since the pixels just occluded by slice *k* would still be considered for slice *k+0.5*. The same also holds true for the completed slice *k+0.5* and slice *k+1*.

To implement intermediate slices, the functions Render(), Composite_Slice(), and Composite_Pixel() need to be modified (see Fig. 8). Before one goes on to change Composite_Pixel() one should realize, however, that pixels that were touched by slice *k* may not always be touched by slice *k+1*, due to the transparency encoding. Hence, performing the compositing only when slice *k+1* is processed will not update these particular pixels, which will lead to wrong rendering results. To make sure that all pixels are updated correctly one needs to composite pixels even for the first part of the intermediate slice, weighted by a factor of 0.5. However, a copy of the original baseplane_image must be saved as well, since in case the second part of the intermediate slice also comes through, one needs to be able to undo the previous compositing and subsequently re-composite with the weighted sum of both parts of the intermediate slice.

Since the sample distance is now half of what is assumed by shear-warp's opacity correction table, we need to normalize the opacity using the following well-known equation [13]:

$$\alpha_n = 1 - \sqrt{1 - \alpha} \tag{6}$$

where $\alpha$ is the sample_opacity in our application and $\alpha_n$ the normalized opacity. Further, if pre-interpolated shading is used, the opacity-weighted colors [29] also need to be normalized. Since we have not seen this published anywhere, we shall briefly derive this result. If we assume a homogenous voxel neighborhood of identical colors $c_i$ and opacities $\alpha_i$ (as was also assumed to derive (6)), then the interpolation result in this voxel neighborhood will yield an opacity-weighted color $c\alpha$. Using these pre-conditions, we can write the composited color of two interpolated, identical opacity-weighted colors as follows:

$$(c\alpha)(1 - \alpha_n) + (c\alpha) = (c\alpha)' \tag{7}$$

To achieve that (cα)' is the same (cα) we would get had we only used the original slices, we need to introduce a normalization factor λ:

$$\lambda(c\alpha)(1 - \alpha_n) + \lambda(c\alpha) = (c\alpha) \tag{8}$$

Solving for λ we get:

$$\lambda = \frac{1}{2 - \alpha_n} = \frac{1}{1 + \sqrt{1 - \alpha}} \tag{9}$$

Multiplying the interpolated opacity-weighted colors with λ yields the desired normalization.

Note that in the special case of a homogenous neighborhood with identical voxel colors $c_i$ and opacities $\alpha_i$ we could also use $c\alpha_n$ instead of $c\alpha$ in (7), in which case no nomalization by λ would be needed. In the general case, however, the $c_i$ and $\alpha_i$ are different, and weighted normalization of the individual voxels $c_i\alpha_i$ would be more costly than normalization of the interpolation result $c\alpha$.

```
Render_Intermediate_Slices (current_RLE)
    base_image.Initialize();
    num_scanline_pixels ← slice_width;
    num_scanlines ← slice_ height;
    for k ← front_slice ... end_slice, +1
        Composite_Slice_Intermediate_Slices (k,0);
        Composite_Slice_Intermediate_Slices(k,0.5);
        Composite_Slice_Intermediate_Slices(k+1,-0.5);
    return(base_image);

CompositeSlice_Intermediate_Slices (k, shear_offset)
    slice_u ← (k+shear_offset) · shear_u + translate_u;
    slice_v ← (k+shear_offset) · shear_v + translate_v;
    ... no further changes

Composite_Pixel_I_S (voxel_square, weights, shear_offset)
if (shear_offset == 0)
        // composite as usual
else if (shear_offset == 0.5)  // 1st half of interm. slice
        // partial compositing, but save previous pixel rgbα
else if (shear_offset == -0.5) // 2nd half of interm. slice
        // final compositing, use partial & prev. pixel rgbα
```

**Figure 8:** *Modified routines for the pseudocode of Fig. 2*

## 4. Extending Shear-Warp to BCC grids

Body-centered grids are an efficient alternative to the traditional cubic cartesian grids we have considered so far. Provided that the signal's frequency spectrum can fit into a spherical shape, one can arrange the spectra into a closer packing - without a loss of spectral information. This is highly desirable since the Fourier scaling theorem states that the closer the spectra are packed in the frequency domain, the further apart samples can be placed in the spatial domain. It turns out that the closest packing for spheres is the face-centered grid (FCC grid), whereas the spatial equivalent of the face-centered grid is the body-centered grid (BCC grid) [6]. See Fig. 9 where we contrast a BCC grid cell with a cubic cartesian one. The samples are spaced further apart in the BCC grid, reducing the number of required samples by 29.3%.

BCC grids have been recently used by Theussl et.al, [26] for Westover-type splatting [28]. In the current work, we have sought to render BCC grids with the shear-warp algorithm. It

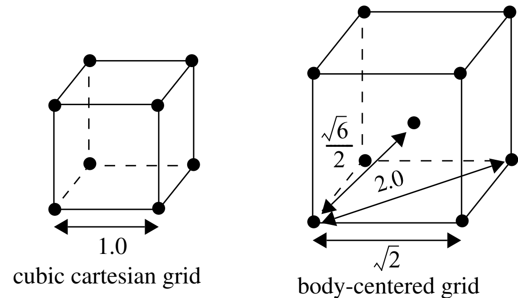



**Figure 9:** *A cubic cartesian grid cell vs. a body-centered grid (BCC) cell, drawn in relative proportions. The BCC grid cell has a sample point in the center of the cell.*

| Dataset (for image/view see Fig. 11 and 12) | | | Matched resolution | | Rendered into a $512^2$ image (with enhancements) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Size | Grid type | Image | S | Magn. | S | I | IP | IM | IMP |
| Cube | $64^3$ | cartesian | $74^2$ | 0.06 | $7^2$ | 0.22 | 0.25 | 0.23 | 1.60 | 1.60 |
| Cube | $91 \cdot 45^2$ | BCC | $74^2$ | 0.04 | $10^2$ | 0.17 | 0.28 | 0.25 | 2.80 | 3.00 |
| Engine | $128^3$ | cartesian | $148^2$ | 0.11 | $4^2$ | 0.30 | 0.40 | 0.38 | 3.30 | 2.90 |
| Engine | $181 \cdot 91^2$ | BCC | $148^2$ | 0.07 | $5^2$ | 0.27 | 0.37 | 0.35 | 3.80 | 4.40 |
| Engine (transparent) | $128^3$ | cartesian | $148^2$ | 0.22 | $4^2$ | 0.44 | - | - | - | 12.0 |
| CT head | $128^3$ | cartesian | $148^2$ | 0.11 | $4^2$ | 0.33 | 0.55 | 0.49 | 4.31 | 4.40 |

**Table 1:** *Timings (in secs) to render the selected datasets (engine, cube, head) for all shear-warp variations presented in this paper. The left part of the table refers to renderings into a window of the same proportions than the dataset. The right part of the table refers to renderings into a window of size $512^2$. The column labeled "Magn." reports the number of samples required per voxel (which were taken when the M option was turned on) to match volume sampling rate with image resolution. (In this table: S: standard shear-warp, I: intermediate slice, P: post-interpolated shading/classification, M: matched. sampling.)*

turns out that the adaptation of shear-warp to handle these grids is quite simple. Once one realizes that the BCC grid can be regarded as two interleaved cubic grids, an elegant solution exists to modify the shear-warp code. The first step is to untangle these two cubic grids at volume read time. Let us assume that both sub-grids have the same extents, just offset by $\sqrt{2}/2$ along all three axes. We can then RLE-encode these two volume separately, using the first gradient estimation method suggested by Theussl et.al. [26], i.e. using voxel neighbors for central differencing that are $\sqrt{2}$ apart. Theussl et.al. also suggest an alternative method that first interpolates samples in the faces of the cube, and then uses these to estimate the gradients. However, we have employed the former method in our final solution since it is more convenient for our purposes. It should also be noted that, although we now have $\sqrt{2}$ more slices than before, the total number of voxels in the runs is reduced since each slice has only 1/2 of the original voxels.

During rendering, we will encounter slices of the two interleaved volumes in an alternating fashion. Hence, we will need to switch back and forth between slices of the two RLE-encoded volumes. But recall that the two volumes are offset by one half the grid distance. Thus, we also need to add a factor of 0.5 to the shear factor for all slices of one of the two volumes. These are all the modifications required, and the modified code is shown in Fig. 10.

```
CompositeSlice_BCC (kk)
    if (Odd (kk))     sub_volume ← 0;
    else              sub_volume ← 1;
    slice_u ← kk · shear_u + translate_u + sub_volume · 0.5;
    slice_v ← kk · shear_v + translate_v + sub_volume · 0.5;
    slice_u_int ← floor (slice_u);
    slice_v_int ← floor (slice_v);
    current_RLE ← current_RLE_array[sub_volume];
    k ← kk / 2;
   ....
```

**Figure 10:** *Modified Composite_Slice() routine to handle BCC grids.*

Note that for shear-warp to work we must assume an in-slice grid spacing of 1.0 in the actual rendering. This means that the volume will appear smaller on the screen if we don't compensate for this by adding a scale factor to the warp matrix. This post-rendering magnification, however, may lead to blurring, and one should really cast $\sqrt{2}$ rays per grid cell. Unfortunately, this is inconvenient for shear-warp, due to its present design constraint that the number of samples per unit must be an integer value. Hence, the only choice at the present time is to use the matched sampling approach outlined earlier and use 4 samples per voxel square, if one seeks to match the resolution of the regular cubic grid.

Extending BCC shear-warp to post-interpolated shading and classification as well as matched sampling is straightforward. The calculation of intermediate slices requires proper shifting of consecutive slices, according to the factors given in Fig. 8 and Fig. 10.

## 5. Results

We incorporated all presented methods into our own implementation of the shear-warp algorithm. At the present time our implementation can only handle orthographic projections. We used a PC with a 1.2 GHz PentiumIII processor and 128 MB of RAM for all experiments. The space-efficient RLE scheme was compared with the traditional one (both using a pointer array to locate individual RLE-encoded scanlines), and we found that both ran at about the same speed.

Table 1 lists the timings and Fig. 11 and Fig. 12 (the colorplate) show the associated images (all $512^2$ pixels). For the labeling, we use the following encodings: S: Standard shear-warp, I: with intermediate slices, P: post-interpolated classification and shading, and M: matched sampling. We then concatenate these to indicate the settings we chose (our application allows us to choose any combination of settings).

The first column in Fig. 11 shows renderings of a binary cube of $64^3$ voxels on a cartesian grid. One observes that pre-interpolated shading and classification causes artifacts on flat

opaque surfaces (S). These artifacts can be reduced somewhat by interpolating intermediate slices (I), but can only be fully removed by first interpolating the voxel densities and then classifying the resulting sample values via the opacity transfer function (IP). However, it is quite noticeable that the cube edges (the sharp object features) still suffer from the undersampling (the image baseplane has $74^2$ pixels, but we need $512^2$). By using $7^2$ samples per voxel square, the cube renders at excellent quality.

The second column in Fig. 11 shows renderings of the binary cube resampled into a BCC grid. In fact, we obtained all BCC volumes from the original cartesian ones via interpolation with a radially symmetric Gaussian filter. The number of samples of the BCC grid is about 72% of its cartesian counterpart. We notice that all renderings have the same quality trends than the cartesian ones, but appear more blurred, which is probably due to the lowpassing of the resampling filter. Although it is true that we could have directly sampled the cube into the BCC grid to reduce the blurring, this cannot be done in the general case since real-life volume datasets do (not yet) come sampled on BCC grids and thus must be resampled anyhow. We shall defer further discussions relating to proper BCC test volumes and filter functions to a forthcoming paper.

In order to evaluate the effectiveness of our modifications in a more formal way, we also rendered the Marschner-Lobb function (MLF) [14]. The MLF is a function that has an almost uniform frequency spectrum for which 99.8% of the frequency content is contained within the Nyquist limit. Setting the isovalue to 128 we observe that only by using post-interpolated classification one can overcome the excessive blurring. Since the MLF has been predominantly used to evaluate interpolation filters with raycasting at a stepsize of 0.1 and less, we cannot expect to get comparable results when using a stepsize of 1.72 with shear-warp. By introducing one intermediate slice per regular slice pair we can improve matters. Ideally one would use a root-finding algorithm to find the exact iso-surface at a density of 128.

Fig. 12 (the colorplate) continues our study with real-life datasets. The two left-most columns show the engine block, viewed along one of the main diagonals (the hardest case for shear-warp). The left-most column shows the volume on the regular cartesian grid, while the center column shows the volume sampled into the BCC grid. We can make similar observations than we did for the cube. Flat surfaces have reduced staircasing artifacts when an intermediate slice is introduced. Using matched sampling improves the rendering quality substantially, especially when post-interpolated classification/shading is used. Again, the BCC grid leads to somewhat blurrier volumes, but otherwise follows the same trends than the cartesian volumes.

The right-most column in Fig. 12 shows the engine rendered in semi-transparent mode. The quality is noticeably improved when all options are turned on. Finally, we also show two renderings of the CT head dataset on a regular Cartesian grid. We observe that post-classified rendering with one intermediate slice can significantly improve rendering quality (note the better specular highlights and fewer aliasing artifacts), at the cost of about 40% more in frame time.

We shall now turn to Table 1, where the first two time-related columns (labeled 'S') indicate that the time required for warping a baseplane image into a larger screen image can be substantially greater than the time spent on the volume rendering. Although this cost could be easily absorbed by employing commodity 2D texture mapping hardware [20], the artifacts due to image-based magnification will still exist, and the following columns of Table 1 focus on our enhancements in these regards. (The timings relate to the images shown in Fig. 11 and 12.) We observe that the intermediate slice interpolation adds about 1/3 of the standard time, while post-classification decreases run-time by a small amount. The cost for interpolating extra samples to match the image resolution is generally more substantial, on an order of magnitude, but depends on data, rendering mode, and magnification.

We also observe in Table 1 that by using the BCC grid we can get speedups of about 33% on average, for the cases in which none of the new options were turned on. This trend reverses when intermediate slices are interpolated, since the BCC grid has more slices and thus requires more intermediate slices to be interpolated. We may add, that for all BCC volumes, in order to compare the timings with respect to equivalent image size, we use the same image resolution than for the cartesian volumes, although the BCC base plane image has a smaller resolution. This means that the displayed image will have to be upsampled by 30% at warp time, a circumstance that also contributes to the blurring observed in the images.

## 6. Conclusions

The primary goal of this paper is to analyze the short-comings of shear-warp, i.e., blurry images on zoom as well as ragged edges and staircase artifacts on flat surfaces, and to devise measures that alleviate these. The former problem can be met by matching the in-slice sampling rate with the image resolution in conjunction with post-interpolation classification and shading, while the latter issue can be helped by interpolating and compositing, on-the-fly, one intermediate volume slice per original slice-pair. While the intermediate slices and the post-classification help improve image quality and can be achieved with only little performance penalty, interpolating extra samples improves the quality most but is expensive and renders the algorithm non-interactive. Thus this can be an option if one wants to quickly switch to a high-quality renderer without leaving the efficient shear-warp volume encoding. One of the major speed gains of shear-warp is rooted in the fact that the volume sampling resolution is independent of the image resolution since the base plane image is simply upsampled for display. As one could do this for any volume rendering algorithm, the increase in runtime for adding extra samples seems justified. As a matter of fact, the runtimes in the IMP columns are quite comparable with frame times achieved by other volume rendering algorithms with good occlusion culling. The fact that the runtime grows at a smaller rate than the number of samples is an indication for the rela-

tive efficiency of the run traversal mechanism.

We also investigated the rendering of volumes sampled into the body-centered (BCC) grid. We found speedups of 33%, on average, paired with a 30% smaller memory footprint for the data. Since shear-warp must use one pixel per voxel-square, the reduced number of slice voxels in the BCC grid translates directly into speed-ups over the cartesian grid algorithm which has more slice voxels. However, we also found that the speedups revert to severe slowdowns once intermediate slices are interpolated, since BCC grids have 42% more slices. The images also tend to be more blurrier than those of the cartesian grid volumes for equivalent image sizes since the BCC grid has 30% less voxels per slice row and column. This can be fixed by using 4 samples per voxel square, which however, brings the complexity above that of cartesian grids and the time advantage is lost.

For future research, it might be interesting to investigate schemes that use four RLE runs on two adjacent slices in conjunction with trilinear interpolation. This would be a consequential improvement along the lines of our intermediate slices. We also would like to improve the calculation of the shading cube by replacing the repeated lighting calculations for each new view point by a simple warp of the shading cube.

**Acknowledgements**

This research was supported, in part, by NSF CAREER grant ACI-0093157. We would like to thank Tom Theussl and Torsten Möller for helpful discussions on BCC grids, and the paper reviewers for their insightful comments.

**References**

[1] http://www-graphics.stanford.edu/software/volpack/

[2] M. Amin, A. Grama, V. Singh, “Fast volume rendering using an efficient, scalable parallel formulation of the shear-warp algorithm,” *Parallel Rendering Symposium’95*, pp. 7-14, 1995.

[3] K. Anagnostou, T. Atherton and A. Waterfall, “4D volume rendering with the Shear Warp factorisation,” *Symp. Volume Visualization and Graphics’00*, pp. 129-137, October, 2000.,

[4] B. Cabral, N. Cam, and J. Foran, “Accelerated volume rendering and tomographic reconstruction using texture mapping hardware”, *Symp. on Volume Visualization’94*, pp. 91-98, 1994.

[5] G. Cameron and P. Undill, “Rendering volumetric medical image data on a SIMD-architecture computer,” *Proc. of Third Eurographics Workshop on Rendering*, pp. 135-145, 1992.

[6] D. Dudgeon and R. Mersereau, *Multi-dimensional Digital Signal Processing*, Prentice-Hall:Englewood Cliffs, 1984.

[7] K. Engel, M. Kraus, and T. Ertl, "High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading," *Proc. SIGGRAPH Graphics Hardware Workshop’01*, pp. 9-16, 2001.

[8] G. Knittel, “The ULTRA VIS system,” *Proc. Volume Visualization and Graphics Symposium*, pp. 71-80, October 2000.

[9] P. Lacroute, *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation,* Ph.D. dissertation, Technical Report CSL-TR-95-678, Stanford University, 1995.

[10] P. Lacroute and M. Levoy, “Fast volume rendering using a shear-warp factorization of the viewing transformation,” *Proc. SIGGRAPH ‘94*, pp. 451- 458, 1994.

[11] P. Lacroute and M. Levoy, “Real-time volume rendering on shared-memory multiprocessors using the shear-warp factorization,” *Proc. Parallel Rendering Symposium’95*, pp. 15-22, 1995.

[12] M. Levoy, “Display of surfaces from volume data”, *IEEE Computer Graphics. & Applications*, vol. 8, no. 5, pp. 29-37, 1988.

[13] B. Lichtenbelt, R. Crane, and S. Naqvi. *Introduction to Volume Rendering,* Prentice-Hall:Saddle River, 1999.

[14] S. Marschner and R. Lobb, “An evaluation of reconstruction filters for volume rendering,” *Proc. Visualization’94,* pp. 100-107, October, 1994.

[15] N. Max, P. Hanrahan, and R. Crawfis, “Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions,” Computer Graphics, vol. 24, no. 5, pp. 27-33, 1990.

[16] M. Meissner, U. Kanus, and W. Strasser, "VIZARD II: A PCICard for Real-Time Volume Rendering", *Proc. Siggraph/Eurographics Workshop on Graphics Hardware’98,* pp. 61--67, 1998.

[17] M. Meissner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis, "A practical comparison of popular volume rendering algorithms," *Proc. 2000 Symp. on Volume Rendering*, pp. 81-90, October 2000.

[18] K. Mueller and R. Crawfis, "Eliminating Popping Artifacts in Sheet Buffer-Based Splatting," *Proc. Visualization’98,* pp. 239-245, 1998.

[19] K. Mueller, T. Möller, and R. Crawfis, "Splatting without the blur," *Proc. Visualization’99*, pp. 363-371, 1999.

[20] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler, “The VolumePro real-time raycasting system,” *Proc. SIGGRAPH 99,* p. 251-260, Los Angeles, CA, August 1999.

[21] C. Resk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl, "Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage-rasterization" *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware’00*, pp. 109-118, 2000.

[22] J. van Scheltinga, J. Smit, and M. Bosma, “Design of an on-chip reflectance map,” *Proc. Eurographics Workshop on Graphics Hardware’95*, pp. 51-55, 1995.

[23] P. Schröder and G. Stoll, “Data parallel volume rendering as line drawing,” *Proc. 1992 Workshop on Volume Visuaization*, pp. 25-32,.

[24] J. Schulze, R. Niemeier, and U. Lang, “The perspective shear-warp algorithm in a virtual environment,” *Proc. Visualization’01*, pp. 207-213, October 2001.

[25] P. Shirley and A. Tuchman. "A polygonal approximation to direct scalar volume rendering," *Computer Graphics*, vol. 24, no. 5, pp. 63-70, (San Diego Workshop on Volume Rendering), 1990.

[26] T. Theussl, T. Möller, and E. Gröller, “Optimal regular volume sampling,” *Proc. Visualization’01*, pp. 91-98, 2001.

[27] H. Tuy and L. Tuy, “Direct 2D display of 3D objects”, *IEEE Computer Graphics & Applications*, vol. 4 no. 10, pp. 29-33, 1984.

[28] L. Westover, “Footprint evaluation for volume rendering”, *Proc. SIGGRAPH’90*, pp. 367-376, 1990.

[29] C. Wittenbrink, T. Malzbender, and M. Gross, “Opacity--weighted color interpolation for volume sampling, “ *Proc. 1998 Volume Visualization Symposium*, pp. 135-142, 1998.

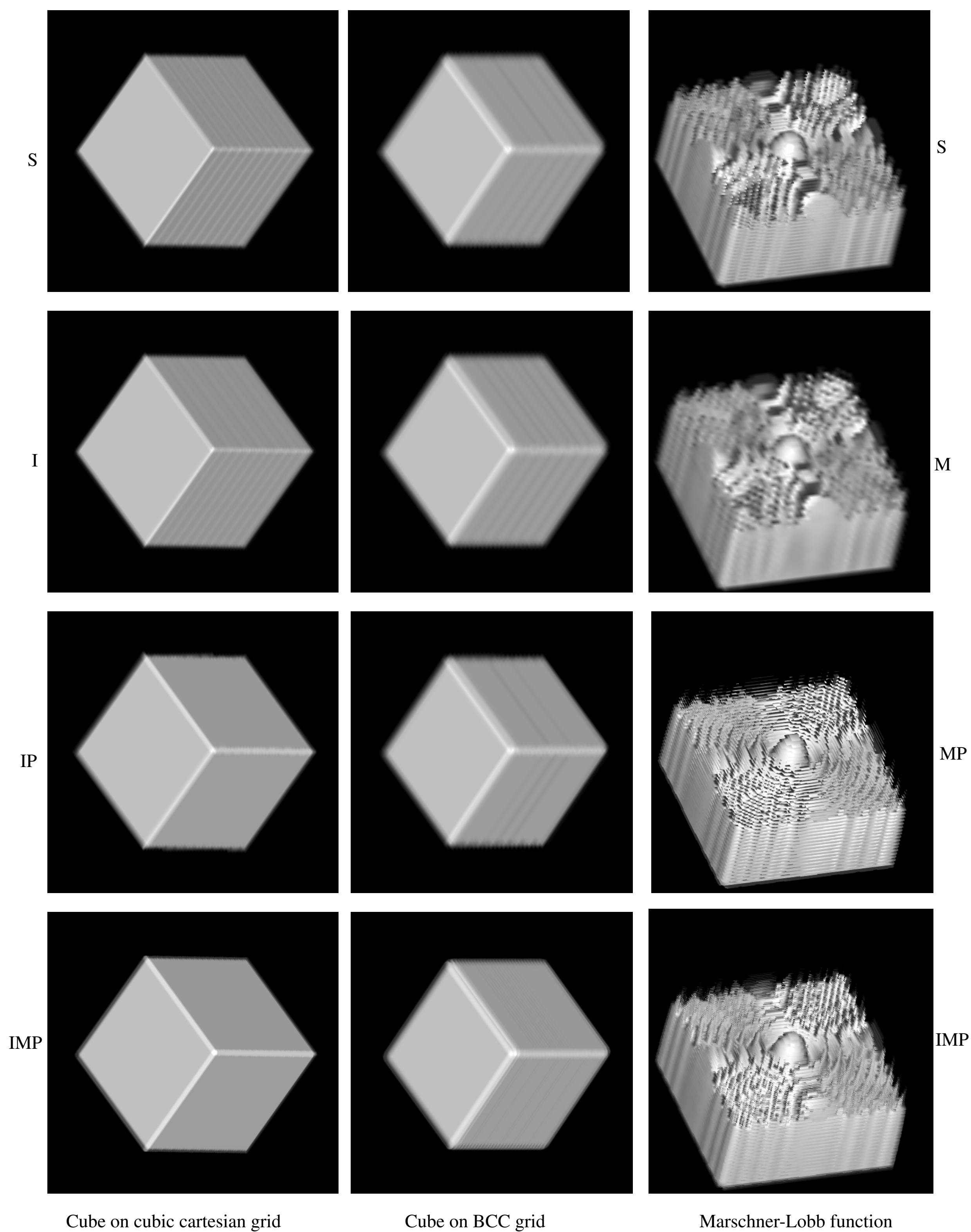



**Figure 11:** *Rendering results, $512^2$ image (S: Standard shear-warp, I: intermediate slice, P: post-interpolated classification, M: matched sampling)*

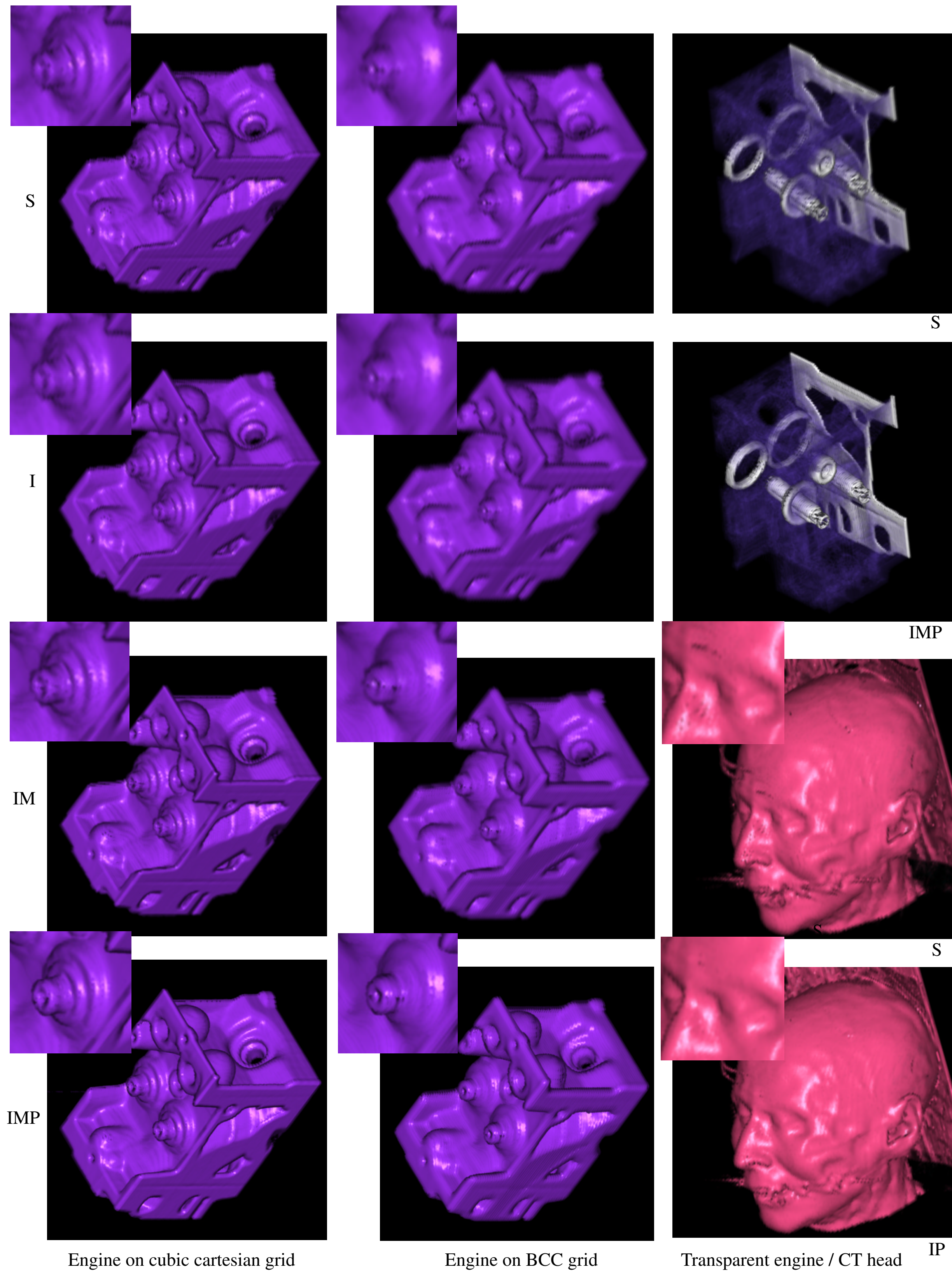



**Figure 12:** *Rendering results, $512^2$ image (S: Standard shear-warp, I: intermediate slice, P: post-interpolated classification, M: matched sampling)*