Visual Comput (2005) 21: 887–896
DOI 10.1007/s00371-005-0298-y

ORIGINAL ARTICLE

Charl P. Botha
Frits H. Post

Improved perspective visibility ordering for object-order volume rendering

Published online: 14 September 2005

C.P. Botha (✉) · F.H. Post
Data Visualisation Group,
Delft University of Technology,
The Netherlands
http://visualisation.tudelft.nl/
c.p.botha@ewi.tudelft.nl

Abstract Finding a correct a priori back-to-front (BTF) visibility ordering for the perspective projection of the voxels of a rectangular volume poses interesting problems. The BTF ordering presented by Frieder et al. [6] and the permuted BTF presented by Westover [14] are correct for parallel projection but not for perspective projection [12]. Swan presented a constructive proof for the correctness of the perspective BTF (PBTF) ordering [12]. This was a significant improvement on the existing orderings. However, his proof assumes that voxel projections are not larger than a pixel, i.e. voxel projections do not overlap in screen space. Very often the voxel projections *do* overlap, e.g. with splatting algorithms. In these cases, the PBTF ordering results in highly visible and characteristic rendering artefacts. In this paper we analyse the PBTF and show why it yields these rendering artefacts. We then present an improved visibility ordering that remedies the artefacts. Our new ordering is as good as the PBTF, but it is also valid for cases where voxel projections are larger than a single pixel, i.e. when voxel projections overlap in screen space. We demonstrate why and how our ordering works at fundamental and implementation levels.

Keywords Volume rendering · Visibility ordering · Splatting

1 Introduction

In volume visualisation, there are three main options: two-dimensional slices of the volume can be rendered, extracted surfaces can be rendered, or direct volume rendering (DVR) can be utilised [3].

Direct volume rendering [5, 7] (DVR) allows the visualisation of structures in the data without having to make decisions about the precise location of object boundaries by extracting polygonal surfaces. Instead, we can define multidimensional transfer functions that assign optical properties to each differential volume element and directly visualise structures on the grounds of this transformation. Most DVR methods can be classified as either image-order or object-order.

Image-order algorithms [7] entail that all pixel locations of the expected result image are traversed. At each pixel location, a view ray is cast through the volume. The volume is sampled at regular positions along this ray. At each position, the scalar volume value is interpolated, very commonly with a trilinear interpolator. These values are transformed by the accompanying transfer function to their corresponding optical characteristics. The order of this operation can be changed: the optical characteristics can be determined before the interpolation stage. All the transformed characteristics along a ray are composited in order to determine the optical characteristics of a single pixel.

In *traditional* object-order volume rendering, one traverses voxel locations instead. At each voxel location, the full contribution of that voxel to the final image is deter-

mined and composited. An important advantage of object-order rendering is that any voxel value is retrieved only once, whereas with ray casting it is most often the case that a single voxel value is sampled multiple times during a single rendering. Ray casting does make early ray termination and frustum clipping optimisations possible. In the same vein, object-order rendering can make use of for instance run length encoding and empty space skipping. However, in order to ensure correct image composition during object-order rendering, one must visit and project the voxels in a strictly front-to-back (FTB) or back-to-front (BTF) order.

Satisfying the ordering constraint means one of the following:

1. The voxels can be sorted according to their distance from the viewpoint before rendering.
2. The inherent regularity of the volume grid can be used to derive a valid a priori visibility order, so that the voxels are retrieved in a correct BTF or FTB ordering.

It is obvious that the latter method, if possible, has the potential to be more efficient, as no sorting operation is required for each rendered view.

For parallel projection, this traversal exists and is easy to find. For perspective projection, the situation is not as positive. The conventional traversals for parallel projection are *incorrect* for perspective projection, as demonstrated by the leftmost image in Fig. 1. Alternative solutions have been proposed, of which the perspective back-to-front ordering, or PBTF, is 'most correct' [12]. PBTF has been proven to be correct for cases where each voxel contributes to at most a single pixel in the resulting image. However, this is most often not the case.

When each voxel affects more than a single pixel, the PBTF ordering introduces very visible rendering artefacts. The centre image in Fig. 1 and the image in Fig. 2 show examples of one such artefact. The 'cross' artefact can easily be seen on the engine block. In the case of the skull, the cross centre is right above its left eye. The skull also shows shading and stairstepping artefacts that are due to incorrect visibility ordering. The reasons for both of these artefacts are explained in Sect. 4.

This is a significant problem, as there is no alternative ordering for perspective object-order volume rendering of discrete voxels.

The research question we attempt to answer in this paper is: *How can we traverse a regularly spaced voxel grid during object-order perspective mode rendering, where voxel projections potentially overlap in screen space, so that a back-to-front ordering with regards to the viewpoint can be derived, without explicitly sorting the voxels?*

This problem is analysed and a solution is proposed. We explain why the PBTF ordering is only partially correct. We then propose a new ordering, called IP-PBTF, that is based on PBTF and rectifies the demonstrated problems. We explain why it works at basic and implementation levels. We also present a method to implement the IP-PBTF ordering that is compatible with empty space-skipping volume rendering implementations. The presented implementation is just as fast as the regular PBTF. An implementation of hardware-accelerated splatting incorporating our new ordering is available.[1]

The rest of this paper is organised as follows. In Sect. 2 we discuss the existing orderings in more detail. We also discuss splatting as one of the better known examples of object-order volume rendering algorithms. The PBTF is explained in Sect. 3, where we also show when, why and how the PBTF fails. We present our new ordering in Sect. 4. In this section we also give implementation-oriented tips and we also show how to implement the ordering for empty space-skipping implementations. Performance timings are given and briefly discussed in Sect. 5. In Sect. 6 we discuss our findings.

2 Related work

It has been shown that, for parallel projection, one can select a traversal direction for each of the axes of the rectilinear grid on which the volume was defined so that the retrieved list of voxels would be ordered strictly back to front with respect to the viewing plane [6]. This BTF ordering is not correct for perspective projection, however. An extreme example of this is shown in the leftmost image in Fig. 1.

Westover's ordering, henceforth called the WBTF as per Swan's exposition [12], added a permutation constraint [14] to the BTF. The permutation constraint determines how the axes should be nested during the traversal: the axis most perpendicular to the view plane is traversed in the outer loop and the axis most parallel to the view plane is traversed in the inner loop.

The WBTF is correct for more cases than the conventional BTF during perspective projection, but it is relatively easy to construct simple examples where it yields incorrect orderings. In practical terms, it yields similar but less severe rendering artefacts than those of the BTF.

Swan unified and proved [12] the perspective-correct orderings previously published by Anderson for 2D [1] and Max for 3D [8]. This order will henceforth be called the PBTF. The PBTF was a significant improvement on the existing orderings and was proved to be correct for cases where voxel projections are no larger than a pixel. However, when this is not the case, the use of the PBTF ordering results in very noticeable rendering artefacts. The centre image in Fig. 1 and the image in Fig. 2 show the very typical PBTF 'cross' artefact.

This is quite a serious problem, as PBTF is currently the only known correct ordering for perspective object-order volume rendering of discrete voxels. In Sect. 3 we

[1] http://cpbotha.net/ShellSplatter

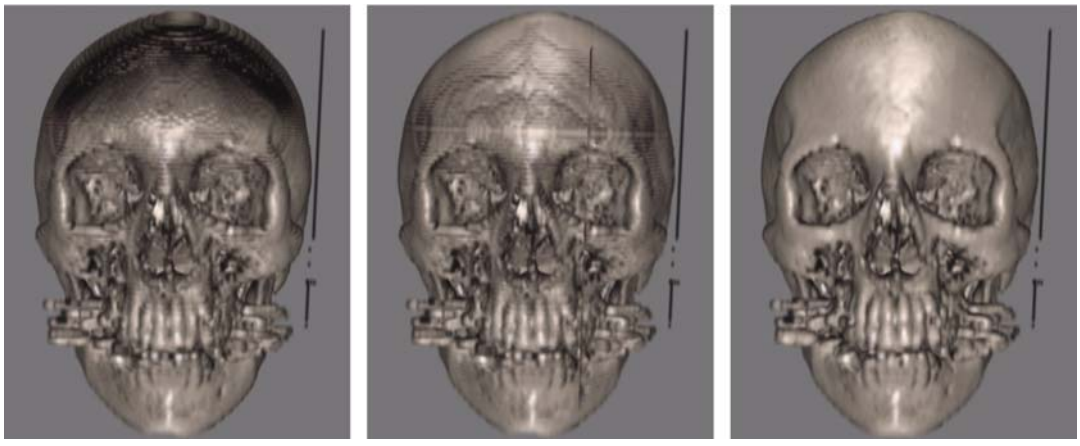

Fig. 1. A perspective splatting of the Stanford CT Head dataset from the same viewpoint with three different orderings: traditional BTF, PBTF and our IP-PBTF

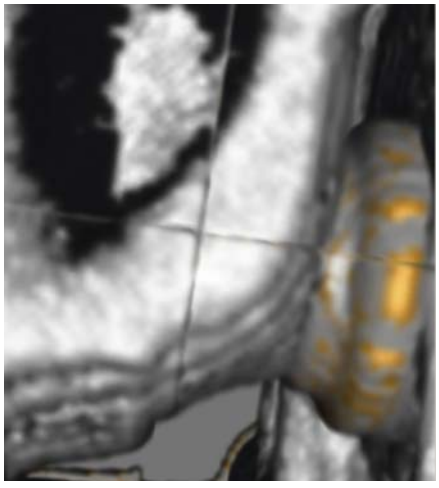

Fig. 2. Close-up of splatting of 'engine block' dataset showing the cross artefact caused by PBTF ordering. The cross artefact occurs at the volume subdivision boundaries due to the fact that splats overlap in screen space

explain the PBTF ordering in more detail and also show why and how it breaks.

In [4], the problem was worked around by adapting the data structures so that the viewing angle was always large relative to the splat size. This minimised the visual artefacts. However, ensuring a large viewing angle in all applications is often not practical. This work also recognised the importance of the traversal axis permutation.

A good summary of direct volume rendering algorithms is available [9]. Splatting is a popular object-order (i.e. forward projection) direct volume rendering method that treats an N-dimensional sampled volume as a grid of overlapping volume reconstruction function kernels, weighted with voxel densities. These weighted kernels, often modelled as Gaussians, are projected onto the image plane to form 'splats' that are composited with affected pixels [13]. In this way, splatting approaches the problems of volume reconstruction and rendering as a single task.

In original splatting, the volume is traversed from front to back or from back to front. Centred at each voxel position is a reconstruction kernel integrated along the view axis to form a pre-integrated 2D kernel footprint. The kernel footprint is used to modulate the looked-up and shaded voxel optical characteristics (colour and opacity) of that voxel and projected onto the image plane where it is composited with the affected pixels. During splatting, a single voxel typically affects more than one pixel, and thus the PBTF ordering is not correct for perspective projection in this case.

The use of pre-integrated reconstruction kernels causes inaccuracies in the composition as each kernel is independently integrated and not in a piecewise fashion, along with other kernels in the path of a view ray. This can result in colour-bleeding of obscured objects in the image. Westover proposed first performing compositing of piecewise kernel integrations into volume axis-aligned sheet buffers and then onto the image buffer to alleviate this effect [14]. However, the axis-aligned sheet buffering resulted in sudden changes in image brightness, also known as 'popping', during rotation. Mueller et al. introduced image-aligned sheet buffers to eliminate this problem [10, 11].

Image-aligned sheet-buffer splatting does not show any of the mentioned artefacts due to the fact that sections of voxel-centred reconstruction functions are accumulated into sheet buffers first. This technique yields the highest quality renderings. However, it is slower and more complex than traditional per-voxel splatting. Also very importantly, per-voxel splatting translates easily to simple and ubiquitous graphics hardware and allows easy hardware-assisted blending with traditional opaque geom-

etry. This is invaluable in, for example, surgical simulation.

In short, traditional per-voxel splatting fills an important niche in the direct volume rendering world. This accentuates the necessity of a correct perspective ordering for the object-order rendering of discrete voxels.

3 PBTF

In this section, we explain the PBTF ordering as proposed by Anderson for 2D [1], Max for 3D [8] and proven by Swan [12]. We also point out, for the first time, the straightforward relationship between the Meshed Polyhedra Visibility Ordering (MPVO) algorithm [15] and the PBTF.

Let $\boldsymbol{v} = (v_x, v_y, v_z)$ be the position of the camera or viewpoint. Depending on the location of $\boldsymbol{v}$ relative to the volume, the volume is partitioned into smaller subvolumes. Very simply speaking, the volume is divided into subvolumes by three dividing planes, each orthogonal to a different volume grid axis and passing through $\boldsymbol{v}$.

For example, if $\boldsymbol{v}$ were completely *inside* the volume, the three dividing planes would divide the volume into eight subvolumes. This is called the 'volume-on' case. If $\boldsymbol{v}$ were to move in the x direction until it was *outside* the volume, the three dividing planes would divide the volume into four subvolumes, as the plane orthogonal to the x-axis passing through $\boldsymbol{v}$ would no longer intersect the volume. This is called the 'face-on' case. If $\boldsymbol{v}$ were now to move in the y direction until it was above the highest extent of the volume, the three dividing planes would divide the volume into two subvolumes, as only the plane orthogonal to the z-axis would still intersect the volume. This is called the 'edge-on' case. Moving $\boldsymbol{v}$ in the z direction until the z-orthogonal plane does not intersect the volume would not result in any partitioning. This is called the 'corner-on' case. Algorithm 1 shows a pseudo-code description of the PBTF partitioning. For each iteration of k, a single axis is split.

Algorithm 1. Determining volume partitioning for the PBTF

Volume dimensions are $(x_0, x_1, y_0, y_1, z_0, z_1)$
$\boldsymbol{v} = (v_x, v_y, v_z)$ is perspective viewpoint
for $k = x, y, z$ **do**
  if $v_k < k_0$ **or** $v_k > k_1$ **then** {v_k is 'outside'}
    $(k_{\min 1}, k_{\max 1}) \Leftarrow (k_0, k_1)$
    $(k_{\min 2}, k_{\max 2}) \Leftarrow (0, 0)$
  else {v_k is 'inside'}
    $k_v \Leftarrow v_k$ rounded to nearest voxel
    $(k_{\min 1}, k_{\max 1}) \Leftarrow (k_0, k_v - 1)$
    $(k_{\min 2}, k_{\max 2}) \Leftarrow (k_v, k_1)$
  end if
end for
output: $(k_{\min 1}, k_{\max 1})$, $(k_{\min 2}, k_{\max 2})$ partitioning for each axis

After having determined the volume partitioning, a traditional BTF traversal is executed for each subvolume, i.e. for each partition an independent set of traversal directions is chosen according to the BTF. Swan's test implementation does indeed include a WBTF per volume subdivision, but his proofs explicitly allow any permutation.

For a given viewpoint, no voxel in any subvolume can occlude any voxel in any other subvolume. This is easy to see, as the planes dividing the volume into subvolumes all pass through the viewpoint. None of the view rays radially emanating from the viewpoint can ever intersect any of the dividing planes, i.e. no view ray can pass through more than a single subvolume. Hence, there can be no occlusion between subvolumes. In other words, we can treat the partitioned subvolumes as independent volumes and render each one separately with a traditional BTF traversal. It is important to note that each partitioned subvolume constitutes a corner-on case.

It is interesting to note that the Meshed Polyhedra Visibily Ordering (MPVO) algorithm proposed by Williams [15] reduces to the PBTF ordering and volume subdivision for a rectilinear volume. Swan mentions this work but does not show the relationship to the PBTF. The MPVO orders the polyhedra in convex meshes by constructing a directed adjacency graph for all polyhedra. A direction is assigned to each edge by constructing a plane through the face separating the two adjacent polyhedra. This plane divides the world into two half-spaces. The direction of the edge is always towards the half-space containing the viewpoint. If a topological sort of this directed adjacency graph is performed, the resultant ordering is a valid BTF order.

The MPVO can be applied to regularly spaced voxel grids if each voxel is seen as a cubic cell. If we were to construct, for each cubic cell, six dividing planes (according to the principle above), there would be a maximum of three main plane orientations. Continuing this reasoning, it is easy to see that in the worst-case scenario (viewpoint within the volume), the volume will be divided into eight regions, where the voxel cuboids in each of the eight regions will have identical graph edge directions. A topological sorting of this graph will result in eight volume subdivisions, each with its own BTF ordering. One can continue this mental exercise for all possible cases (volume-on, face-on, edge-on and corner-on), and the MPVO will reduce to the PBTF ordering every time.

Note that the MPVO, just like the PBTF, only determines different sets of traversal directions for all subvolumes but places no constraint on the axis permutation of each subvolume. Any of the three permutations (for any of the subvolumes) constitutes a correct topological sort of the directed adjacency graph.

4 IP-PBTF

By showing a few simple examples, we demonstrate the two major reasons why PBTF generates incorrect orderings. Identifying the deficiencies leads directly to the remedies. Subsequently, we analyse the problem in more detail and show that our ordering is an improvement in all possible cases. We then present some implementation details.

4.1 Constructing the IP-PBTF

When voxel projections are no larger than a single pixel, there is generally no occlusion between subvolumes. However, when voxel projections are larger than a single pixel, this assumption is incorrect, i.e. significant occlusion takes place between subvolumes. This is the primary reason why the PBTF ordering is incorrect for the case where voxel projections are larger than a pixel. Figure 3 shows a very simple 2D example where the PBTF is not sufficient. Notice that, due to the larger-than-pixel voxel projections, voxel 2 is composited *before* voxel 4, and voxel 3 is composited *before* voxels 4 and 5. In a larger volume, this incorrect compositing will take place along any subvolume division and cause the visible artefacts that we have shown. The cross artefact in Fig. 2 is a specific example of this.

Studying the problem at this scale already presents the first part of our solution. By interleaving the voxels along the volume division, i.e. rendering voxels alternatingly from the top and bottom subvolume whilst still maintaining the intrasubvolume BTF ordering, we would solve at least the problem shown in Fig. 3. Figure 4 shows the results of this change.

In this simple case, the interleaving seems to solve the visibility ordering. However, we can construct another simple case that shows how the interleaving by itself is not quite sufficient. In Fig. 5 interleaving has been applied: the voxels are rendered alternatingly from subvolumes 1 and 2. However, the slowest changing grid axis index belongs to the axis orthogonal to the subvolume

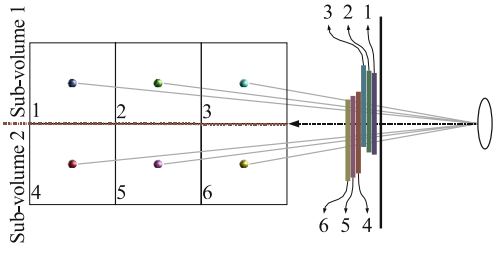

Fig. 3. A simple 2D example showing how the PBTF visibility ordering is incorrect for cases where the voxel projection is larger than a single pixel. Each *numbered block* represents a voxel

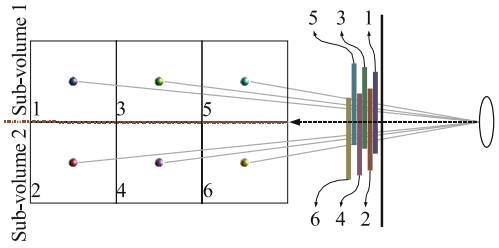

Fig. 4. Interleaving the ordering of the voxels in the subvolumes solves the problem shown in Fig. 3

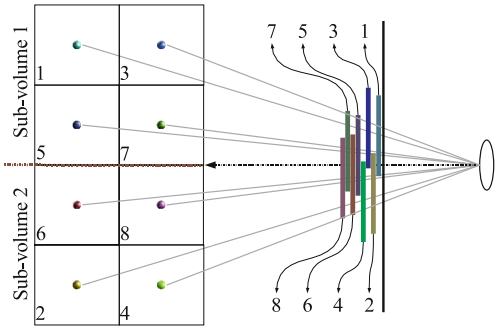

Fig. 5. Subvolume interleaving by itself is not sufficient to remedy the PBTF visibility problems. In this simple case, in spite of interleaving, voxels 3 and 4 have been incorrectly composited before voxels 5 and 6

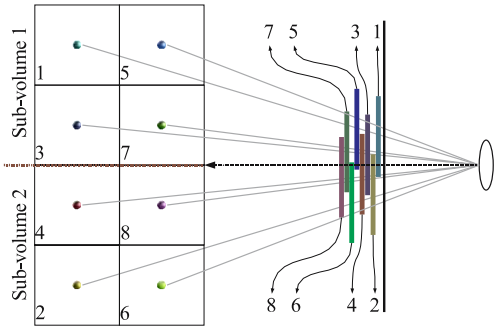

Fig. 6. Interleaving between subvolumes *and* selecting a suitable axis permutation remedies the PBTF problems in this example

division. In this very simple case, voxels 3 and 4 are incorrectly rendered *before* voxels 5 and 6. Once again, in a more extensive volume, this will cause visible artefacts.

From this example, it is clear that one should carefully choose an applicable permutation: the slowest changing axis index should never be the one belonging to the axis orthogonal to any of the subvolume division planes. The result of applying this rule to the simple example in Fig. 5 is shown in Fig. 6. Note that all voxels are now projected in the correct ordering. We reiterate

the fact that Swan's proof allows any axis permutation for the PBTF subvolumes, although his specific implementation did include a WBTF ordering for each subvolume.

4.2 Analysis

Our new ordering for perspective object-order volume rendering is called the Interleaved and Permuted Perspective Back-to-Front ordering, or IP-PBTF. We have demonstrated with a few simple examples why the PBTF is not correct for cases where each voxel affects more than one pixel and how the IP-PBTF would be correct in these cases as well. There are two reasons why the PBTF is incorrect for these cases:

1. There is a definite interaction between subvolumes.
2. A non-permuted BTF (i.e. traditional) ordering is often incorrect even for a simple corner-on case, which is what every subvolume reduces to in the PBTF ordering.

Studying Fig. 7, which is valid for every possible 2D perspective projection configuration and can be constructed for the 3D case, it becomes obvious why the PBTF breaks. From this figure, it is clear that rendering for example voxels 2, 3 and 4 before 1 would be incorrect, as they are inside the outermost circle and 1 is outside, indicating that 1 is further away from the viewpoint and should be rendered first. This is exactly what the PBTF ordering would do, as it would completely render subvolume 1 (SV 1) before subvolume 2 (SV 2). Interleaving the two subvolumes would order these correctly.

Interestingly, this figure also indicates the necessity for a *PBTF-based* ordering, i.e. one where the volume is partitioned into viewpoint-related subvolumes. If we were to attempt a WBTF ordering for the whole volume, x would be chosen as the slowest changing axis, i.e. the outer loop, and y as the fastest changing, i.e. the inner loop. However, the direction of y would be either $0 \rightarrow y_1$ or $y_1 \rightarrow 0$, neither of which can potentially yield a purely BTF ordering, as can be seen from the outermost circle: as we move outwards from the volume division line, the distance increases! This clearly indicates that the volume should be split and that the subvolumes should be traversed in opposite directions of y, *towards* the subdivision line (plane in 3D).

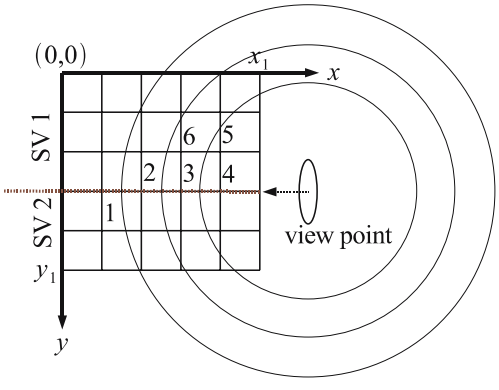

Fig. 7. Circles (spheres in 3D) of equidistance from the viewpoint. It is clear that the y-axis should be partitioned and that each partition should be traversed in the opposite direction in order to maintain a BTF ordering. This illustration also helps to show why interleaving and permutation are required. See the text for more detail. 'SV 1' and 'SV 2' refer to 'subvolume 1' and 'subvolume 2' respectively

The figure also shows why the traversal axes of each subvolume should be permuted so that the slowest changing axis is not orthogonal to the volume subdivision line (plane in 3D). If we were to traverse subvolume 1 with the y-axis as the slowest changing, we would render for example voxel 5 *before* voxel 3 and all voxels to its left, voxel 6 *before* voxel 2 and all voxels to its left, and so forth. This is of course not a correct ordering.

The permutation constraint cannot be satisfied for the volume-on case, i.e. the case where the viewpoint is located *inside* the volume that is being rendered. In the volume-on case, there are three volume subdivision planes and eight subvolumes. There are three volume axes, i.e. one orthogonal to each subdivision plane, which means that it is impossible to select a slowest traversal axis that is not orthogonal to a division plane. In this case, a good practical solution is to select the axis which is most perpendicular to the view plane as the slowest changing axis.

This also brings us to a limitation of almost all current object-order direct volume rendering methods: Distance from the viewpoint increases radially outwards from the viewpoint, i.e. the distance field is spherical, but grid-traversal orderings are constrained to the orthogonal volume grid. Thus, there are still bound to be cases when voxel pairs are incorrectly ordered, especially with very large voxel projections. In other words, if there were no constraints on processing speed, one would employ extremely thin and spherical sheet buffers with the viewpoint as centre. If there were no constraints on processing speed, but one had to render a voxel at a time, the distance measure around the viewpoint could be sampled at the voxel centres and the voxels rendered in order of decreasing distance value.

That being said, we believe that the IP-PBTF is currently the best grid-constrained ordering for the object-order perspective direct volume rendering of discrete voxels. It is applicable to all cases where a volume on a rectilinear grid is volume rendered with discrete voxel projections overlapping in screen space and where the cost of explicit sorting is not affordable.

4.3 Implementing an interleaved split-dimension traversal

The IP-PBTF ordering is conceptually very simple. However, creating an efficient implementation is rather complex. Subvolumes often have differing sizes on all dimensions as each dimension is independently split. The interleaving is not trivial, as the split most often yields two parts of differing size, which means that the interleaving only takes place part of the time. In addition, each subvolume has its own set of traversal directions.

The crux of a good implementation is an efficient split-dimension traversal, i.e. code to handle the interleaving of the various axes. In order to facilitate implementation, we show an efficient way of setting up and executing a single split dimension. Algorithm 2 shows how to set up a single split dimension. This has to be done for all dimensions that have been partitioned by any of the dividing planes. Fortunately, this part happens only once per frame rendered. Algorithm 3 shows how to iterate through such a split dimension during the actual voxel projection phase.

An axis is partitioned into two sections by the split point k_v. The largest section is determined and its endpoints are stored in b_0 and b_1. b_i determines the direction with which we traverse the largest section. We also determine a threshold, b_t. When the larger section index, b, reaches this threshold, the interleaving with the smaller dimension starts. s is used as an index for traversing the smaller section of the axis. Its direction is always opposite to that of the b index.

If the currently traversed interleaved axis has nested axes, *nestedLoop()* is called twice. Each of the *nestedLoop()* calls represents another potentially split dimension, but could also represent a straightforward non-interleaved axis traversal, depending on the relevant PBTF partitioning case. If the axis that is being traversed is already at the most nested level, *visitVoxel()* is called, meaning that that particular voxel is rendered. For three nested split dimensions for example, this means that *visitVoxel()* will be called a maximum of eight times during each iteration of the outer loop.

Algorithm 2. Setting up a split dimension. See Sect. 4.3 for details.

```
split-dimension is k0 ≤ k ≤ k1
split-point is kv
if kv − k0 > k1 − kv then
  b0 = k0
  b1 = kv + 1
  bi = 1
  bt = (kv − k0) − (k1 − kv)
  s0 = k1
else
  b0 = k1
  b1 = kv
  bi = −1
  bt = (k1 − kv) − (kv − k0)
  s0 = k0
end if
```

Algorithm 3. Iterating through a single split dimension. See Sect. 4.3 for details.

```
s = s0
if s = bt then
  interleaved = true
else
  interleaved = false
end if
if atMostNestedLevel then
  action = visitVoxel
else
  action = nestedLoop
end if
for b = b0; b ≠ b1; b = b + bi do
  action(b)
  if interleaved then
    action(s)
    s = s − bi
  else
    if b + bi = bt then
      interleaved = true
    end if
  end if
end for
```

4.4 Efficient interleaving with space skipping

If the volume rendering algorithm allows random access to the voxels without a performance penalty, the interleaved split-dimension traversal described above is sufficient and should not affect performance significantly.

However, many voxel-based volume rendering algorithms make use of some form of empty space skipping. In general, only voxels that will actually contribute to the rendering are compactly stored and other voxels are completely ignored. Because the significant voxels account for a very small percentage of the complete volume and no time is wasted even examining non-significant voxels, these schemes result in significantly faster rendering.

The interleaved split-dimension scheme detailed above performs explicit interleaving, i.e. we have counter variables iterating through all dimensions to ensure that the interleaving is correctly done. With many empty space skipping implementations, such an explicit interleaved traversal would partly negate the advantages of the space skipping, as non-significant voxel positions would still be traversed by counter variables in order to ensure a correct interleaving. Even if a particular voxel in one subvolume is non-significant, the voxels at matching positions in other subvolumes have to be rendered in an interleaved fashion.

For cases such as these, an implicit interleaving traversal is more appropriate. We have dubbed it ‘implicit’ as no counter or traversal variable is involved. In this section we present a simple technique for implicit interleaving and, as we show in Sect. 5, the implicit interleaving allows us to perform empty space skipping whilst correctly interleaving, without a measurable speed difference compared to the regular PBTF.

For any given axis permutation, we make use of space skipping only for the inner loop, i.e. the fastest changing dimension. Consequently, we only make use of implicit interleaving for the fastest changing dimension. For the outer loops, i.e. the slower changing dimensions, we make use of the standard explicit interleaving scheme described in Sect. 4.3.

Algorithm 4 shows the implicit interleaving technique over one complete inner loop. This is done for each combination of the two explicitly interleaved or non-interleaved outer-loop counter variables.

At the start of the inner loop, the relevant space skipping voxel run for each partitioned subvolume is determined and a pointer for each run is set to the start or the end of that run depending on the PBTF-required traversal direction. For each of these pointers, the distance to the partitioning plane orthogonal to the space skipping dimension is calculated by a simple subtraction.

The greatest distance is determined. All current voxel run pointers that represent a voxel with distance equal to this greatest distance are rendered and then incremented or decremented depending on the various required traversal directions. These directions are always towards the partitioning plane, so the distances and the maximum distance are by definition decreasing. A new distance is calculated for each incremented or decremented voxel. If a pointer is incremented or decremented and the voxel that it represents subsequently crosses the partitioning plane, that pointer is deactivated. After rendering, incrementing and calculating new distances for the relevant voxels, we start again by determining the new maximum distance. The inner loop is terminated when all voxel pointers have been deactivated.

Algorithm 4. The complete implicit interleaving inner loop.

```
N ⇐ number of subvolumes (maximum 8)
for i = 0 to N − 1 do
  p_i ⇐ initial voxel pointer for subvolume i
  if IsValid(p_i) then
    d_i ⇐ distance of voxel p_i from subdivision
    Δp_i ⇐ increment and direction for subvolume i
  else
    Deactivate(p_i)
  end if
end for
while IsActive(p_0) or IsActive(p_1) or ... or IsActive(p_{N−1}) do
  d_max = max({d_0, d_1, ..., d_{N−1}}) {only for active p_i's}
  for i = 0 to N − 1 do
    if IsActive(p_i) and d_i = d_max then
      renderVoxel(p_i)
      p_i = p_i + Δp_i
      d_i ⇐ distance of voxel p_i from subdivision
      if d_i < 0 then {p_i has skipped across the subdivision}
        Deactivate(p_i)
      end if
    end if
  end for
end while
```

This scheme is simple to implement and makes efficient use of the space skipping encoding whilst implicitly interleaving voxels. In principle, it should work with any space skipping technique where the IP-PBTF is applicable.

5 Results

We have implemented the IP-PBTF ordering as part of a specialised hardware-accelerated splatting implementation called 'ShellSplatting' [2]. The IP-PBTF ordering is applicable to *any* object-order perspective volume rendering implementation where discrete voxels are projected.

Figure 8 shows the same rendering as in Fig. 2, but with the IP-PBTF applied. The cross artefact has clearly disappeared. Figure 9 shows two extreme close-up renderings of the engine block with all splats rendered as circles instead of preintegrated Gaussians in order to emphasize the nature of the cross artefact and the working of the IP-PBTF ordering.

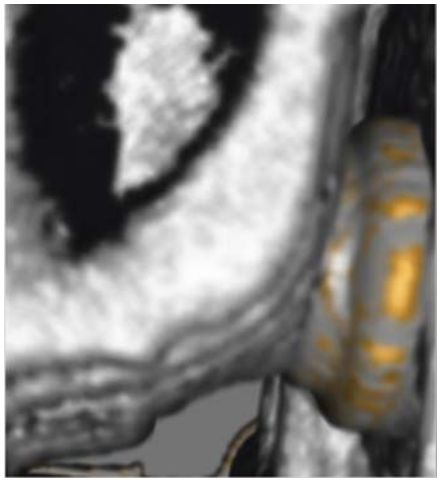

Fig. 8. The same rendering as in Fig. 2, but with the IP-PBTF ordering applied. The cross artefact has disappeared. In this case, the permutation was already correct; only the interleaving had to be applied

Table 1 shows some speed timings for three well-known datasets.[2] These timings were performed on a 2.4 GHz Pentium 4 with a GeForce4 graphics card. The image size was 512×512, and we zoomed in so that the image was generously filled with the rendered volume. We tested BTF, PBTF and IP-PBTF orderings. The first two were unpermuted. The BTF was of course tested in an orthogonal projection setting. A sequence of 3610 different frames was rendered three times per ordering for

[2] The aneurism data are courtesy of Philips Research, Hamburg, and the engine block is courtesy of General Electric. Both were downloaded from http://volvis.org/. The CT Head is courtesy of Stanford University

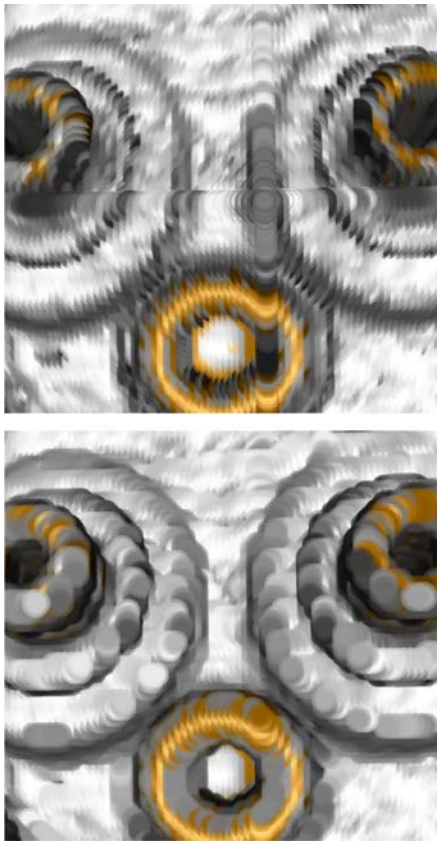

Fig. 9. Two close-up renderings of the engine block dataset with circular splats to illustrate the nature of the cross artefact and the IP-PBTF's remedy. In the *upper image* a PBTF visibility ordering was used, and in the *lower image* the IP-PBTF ordering was used. The permutation has been corrected and the interleaving has been applied

Table 1. Optimised splatter rendering frame rates for the different orderings for a 512×512 rendered image. The IP-PBTF is just as fast as the PBTF but yields a more correct ordering

	Frame rate (frames per second)		
Ordering	Aneurism 256^3	CT Head $256^2 \times 113$	Engine Block $256^2 \times 128$
BTF	49	12	12
PBTF	45	12	13
IP-PBTF	45	12	13

all datasets, i.e. the complete sequence was run 27 times in total. For each ordering and dataset permutation, the average of the three tests was taken and then this average was rounded down to yield the final performance figure in frames per second.

The results show that the IP-PBTF extension has no measurable impact on the rendering performance of our splatting implementation. In other words, the IP-PBTF is just as fast as the PBTF *and* yields higher quality results. Our implementation utilises empty space skipping, so we have used implicit interleaving for the inner loop as discussed in Sect. 4.4. Experiments with *explicit* interleaving in the inner loop resulted in 57% slower rendering for the 'aneurism' dataset. From this we conclude that implicit interleaving is also crucial for an efficient implementation of the IP-PBTF that is compatible with empty space skipping. For the 'CT Head' and 'Engine Block' datasets, which have significantly less empty space than 'aneurism', the difference is much smaller, but implicit interleaving still yields the most efficient implementation.

6 Conclusions

In this paper, we have discussed existing visibility orderings for object-order volume rendering based on discrete voxel projection. We have shown that, for the common case where voxel projections are larger than a single pixel, *none* of the existing orderings yields a correct BTF ordering for perspective projection.

We focused on the best of the existing orderings, namely the PBTF, and showed how and why it is not valid for cases where each projected voxel affects more than a single pixel. By studying the nature of its failure on constructed simple examples, we were able to determine the changes necesary to remedy these failures. The PBTF fails because it assumes that there is no interaction between the partitioned subvolumes. By visiting all voxels in an interleaved fashion, i.e. alternatingly from all subvolumes and by choosing a correct WBTF-like traversal for each subvolume, the PBTF problems are remedied. We have dubbed the new ordering IP-PBTF, or Interleaved and Permuted Perspective Back-to-Front.

We analysed our changes by looking at circles of equidistance (spheres in 3D) with the perspective viewpoint as centre. This showed why a permuted interleaving of the PBTF was necessary. Coincidentally, it is also an effective way of illustrating the necessity of partitioning the volume into subvolumes.

We then presented a possible implementation of iteration through an interleaved split dimension. We also showed how to implement interleaving efficiently for empty space skipping implementations. The former technique is called explicit interleaving and the latter implicit interleaving.

We presented comparative timings of the BTF, PBTF and IP-PBTF orderings in an accelerated splatting implementation. The results show that the IP-PBTF is just as fast as the PBTF for our empty space skipping implementation, if explicit interleaving is used for the outer loops and implicit interleaving for the inner loop.

For projections no larger than a pixel, IP-PBTF is *as good* as the traditional PBTF, as the interleaving and permutation still maintain the PBTF ordering as well. For

larger-than-pixel voxel projections, it is clear that interleaving and permuting properly ensures a more strict BTF ordering during object-order perspective projection of discrete voxels, without explicit sorting. This observation, and the fact that these changes have no measurable impact on performance, lead us to conclude that the IP-PBTF is definitely to be preferred for the perspective object-order volume rendering of discrete voxels.

Acknowledgement This research is part of the DIPEX (Development of Improved endo-Prostheses for the upper EXtremities) program of the Delft Interfaculty Research Center on Medical Engineering (DIOC-9).

We would like to thank Dr. J. Edward Swan for enlightening and pleasant conversations and for his exceptionally clear exposition of PBTF and its proof.

Jorik Blaas can always be relied on to poke holes in one's theories.

References

1. Anderson DP (1982) Hidden line elimination in projected grid surfaces. ACM Trans Graph 1(4):274–288
2. Botha CP, Post FH (2003) ShellSplatting: interactive rendering of anisotropic volumes. In: Bonneau GP, Hahmann S, Hansen CD (eds) Data Visualization 2003 (Proceedings of Joint Eurographics – IEEE TCVG symposium on visualization), pp 105–112
3. Brodlie K, Wood J (2000) Recent advances in visualization of volumetric data. In: Eurographics state of the art reports, pp 65–84
4. Coconu L, Hege HC (2002) Hardware-accelerated point-based rendering of complex scenes. In: Proceedings of the 13th Eurographics workshop on rendering. Eurographics Association, pp 43–52
5. Drebin RA, Carpenter L, Hanrahan P (1988) Volume rendering. In: Proceedings of SIGGRAPH '88. ACM Press, New York, pp 65–74. DOI http://doi.acm.org/10.1145/54852.378484
6. Frieder G, Gordon D, Reynolds RA (1985) Back-to-front display of voxel-based objects. IEEE Comput Graph Appl 5(1):52–60
7. Levoy M (1988) Display of surfaces from volume data. IEEE Comput Graph Appl 8(3):29–37
8. Max NL (1993) Sorting for polyhedron compositing. In: Hagen H, Müller H, Nielson G (eds) Focus on scientific visualization. Springer, Berlin Heidelberg New York, pp 259–268
9. Meissner M, Huang J, Bartz D, Mueller K, Crawfis R (2000) A practical evaluation of popular volume rendering algorithms. In: Proceedings of the syposium on volume visualization and graphics, pp 81–90
10. Mueller K, Crawfis R (1998) Eliminating popping artifacts in sheet buffer-based splatting. In: Proceedings of IEEE Visualization '98, pp 239–245
11. Mueller K, Shareef N, Huang J, Crawfis R (1999) High-quality splatting on rectilinear grids with efficient culling of occluded voxels. Trans Visualizat Comput Graph 5(2):116–134
12. Swan JE (1998) Object-order rendering of discrete objects. PhD thesis, The Ohio State University, Columbus, OH
13. Westover L (1989) Interactive volume rendering. In: Proceedings of the Chapel Hill workshop on volume visualization. ACM Press, New York, pp 9–16. DOI http://doi.acm.org/10.1145/329129.329138
14. Westover L (1990) Footprint evaluation for volume rendering. In: Proceedings of SIGGRAPH '90. ACM Press, New York, pp 367–376
15. Williams PL (1992) Visibility-ordering meshed polyhedra. ACM Trans Graph 11(2):103–126

CHARL P. BOTHA recently completed his Ph.D. research on architectures and techniques for medical visualisation and image processing with the primary goal of applying these techniques to the shoulder replacement process.
He is currently working as a postdoc in the Data Visualisation research group at the Delft University of Technology and more specifically within the Virtual Laboratory for e-Science (VL-e) research programme. His primary focus is on investigating tools and techniques for medical visualisation in a grid-based environment.

FRITS H. POST is an associate professor of computer science (visualisation) at the Delft University of Technology, The Netherlands, where he leads a research group in data visualisation since 1990. His current research interests include volume visualisation, vector field and flow visualisation, medical imaging and visualisation, virtual reality, and interactive exploration of large time-varying datasets.
He is currently chair of the Eurographics Working Group on Data Visualisation and a co-founder of the annual joint Eurographics-IEEE VGTC EuroVis Symposium. He is a member of Eurographics, IEEE Computer Society, and ACM Siggraph.