Volumetric Attribute Filtering and Interactive Visualization Using the Max-Tree Representation

Michel A. Westenberg, Jos B. T. M. Roerdink, Senior Member, IEEE, and Michael H. F. Wilkinson, Senior Member, IEEE

Abstract—The Max-Tree designed for morphological attribute filtering in image processing, is a data structure in which the nodes represent connected components for all threshold levels in a data set. Attribute filters compute some attribute describing the shape or size of each connected component and then decide which components to keep or to discard. In this paper, we augment the basic Max-Tree data structure such that interactive volumetric filtering and visualization becomes possible. We introduce extensions that allow 1) direct, splatting-based, volume rendering; 2) representation of the Max-Tree on graphics hardware; and 3) fast active cell selection for isosurface generation. In all three cases, we can use the Max-Tree representation for visualization directly, without needing to reconstruct the volumetric data explicitly. We show that both filtering and visualization can be performed at interactive frame rates, ranging between 2.4 and 32 frames per seconds. In contrast, a standard texture-based volume visualization method manages only between 0.5 and 1.8 frames per second. For isovalue browsing, the experimental results show that the performance is comparable to the performance of an interval tree, where our method has the advantage that both filter threshold browsing and isolevel browsing are fast. It is shown that the methods using graphics hardware can be extended to other connected filters.

Index Terms—Connected filters, mathematical morphology, Max-Tree, nonlinear filtering, volume visualization.

I. INTRODUCTION

Attribute filters are a subset of morphological connected filters, a class of shape-preserving filters used in image processing [1]–[8]. Examples of such filters are area openings and closings, which remove image detail smaller than a particular area [9]. More general are the attribute openings, which accept or reject image details based on size criteria [10]. Similarly, it is also possible to filter on shape, rather than on size criteria. This idea has been formalized as so-called shape filters [11], [12], which have been extended to 3-D and have been applied to the problem of vessel enhancement in angiographic volume data sets [13].

These filters are different from the morphological erosion- and dilation-based operations in that they do not make use of structuring elements, but rather retain or discard connected components as a whole. An example of an attribute filter applied to a grey scale volume is shown in Fig. 1. The left image shows an isosurface visualization of the original data, and the right image shows an isosurface visualization of the result of filtering the data with an attribute filter that is selective for noncompact structures [13]. For both images, the same isovalue $t = 13$ was used. A computationally efficient implementation of these filters is based on a data structure called the Max-Tree [14].

In previous work, we have developed a volumetric filtering method based on attribute filters, and we have combined it with a basic volume visualization method [15]. In that work, we have shown that filtering can be done at interactive rates; however, this was not the case for visualization. The reason is that we first reconstructed the filtered volume data, and then applied a visualization method. The purpose of this paper is to extend the Max-Tree data structure, such that filtering and visualization can be done directly from this extended Max-Tree. We will show that this can be done at interactive rates. Since there is often no a priori “right” filter threshold, such interactive feedback is essential. As visualization methods, we employ both direct volume rendering and isosurface rendering. The direct volume rendering approaches most useful for our application are splatting [16] and 3-D hardware texture mapping [17]. For isosurface visualization, the extensions proposed in this paper allow fast selection of cells that intersect the isosurface (active cells) when varying the attribute filter threshold and also when varying the isovalue threshold.

The organization of the paper is as follows. Section II describes previous work on attribute filters, the Max-Tree data structure, and volume-rendering methods. We introduce three new attributes in Section III. In Section IV, we introduce the augmented Max-Tree and the visualization methods that make use of the augmented data structure. We compare the results of our new method with that of some standard approaches in Section V. Finally, Section VI discusses the results and possible future extensions.
II. PREVIOUS WORK

A. Attribute Filters and the Max-Tree

An efficient implementation of attribute filters relies on computing both the hierarchy of connected components in the data set and some attribute for each component. The Max-Tree was introduced to separate the computation of these from the actual filtering process [14].

Max-Trees are closely related to contour trees [18]–[21]. In particular, the join trees used to compute contour trees are essentially the same as Max-Trees, especially when they are augmented to include attribute information in each node [22]. A key difference between the Max-Tree and the join tree is that, in the latter case, it is usually assumed that all nodes in the grid of data have a unique (grey) value. By contrast, Max-Trees were designed to administrate flat zones, i.e., connected sets of vertices with the same grey value. Contour trees themselves are equivalent to level-line trees [23], which can be constructed from a Max-tree and a Min-Tree of the same image.

1) Construction: Let $\mathbf{M} \subseteq \mathbb{R}^3$ be the volume domain, and $f : \mathbf{M} \rightarrow \mathbb{R}$ the grey scale volume. We implicitly assume the existence of a grid on $\mathbf{M}$, and we use a 26-connected voxel neighborhood.

A Max-Tree is a tree where the nodes represent sets of flat zones of $f$. A set $F \subseteq \mathbf{M}$ is called a flat zone if for all $p, q \in F$ there exists a path from $p$ to $q$ along which the function value is constant, and the set $F$ is maximal in size. The threshold set $X_h(f)$ of $f$ is the set of points that remain after thresholding at level $h$, i.e.,

$$X_h(f) = \{x \in \mathbf{M} | f(x) \geq h\}.$$  

A peak component $P_h^k$ at a grey level $h$ can then be defined as the $k$th connected component of the threshold set $X_h(f)$. A Max-Tree node $C_h^k$ represents one or more flat zones at grey level $h$ contained in a single peak component at the same level. A simple 1-D example for a signal with nine elements is shown in Fig. 2(a). The signal also has nine flat zones, since none of the signal elements is connected to another element of the same value. The values range between zero and three; therefore, there are four threshold sets, of which only $X_2(f)$ contains two connected components.

The Max-Tree can be constructed by a recursive flooding procedure that makes use of a hierarchical FIFO queue to process the voxels in correct order [14] or by relying on Tarjan’s union-find algorithm [24], [25]. In this paper, we use the queue-based approach.

2) Filtering: Filtering removes Max-Tree nodes based on some property defined by an attribute value $A(P_h^k)$ of a node $C_h^k$ from a universe (typically, $\mathbb{R}$ or $\mathbb{Z}$) on which an order $\leq$ exists. Given a threshold value $\lambda$ from this universe, the algorithm decides whether to preserve, or remove a node. Two classes of strategies exist 1) pruning strategies, which remove all descendants of $C_h^k$ if $C_h^k$ is removed; and 2) nonpruning strategies, in which the parent pointers of children of $C_h^k$ are updated to point to the oldest “surviving” ancestor of $C_h^k$.

Salembier describes four different rules for the algorithm to filter the tree: the Min, the Max, the Viterbi, and the Direct decision. The first three are pruning strategies. In addition, Wilkinson and Urbach [11] introduced another nonpruning strategy, called the Subtractive decision. The decisions of these rules are as follows.

- **Min**: A node $C_h^k$ is removed if $A(P_h^k) < \lambda$ and if one of its ancestors is removed.
- **Max**: A node $C_h^k$ is removed if $A(P_h^k) < \lambda$ and all of its descendant nodes are removed, as well.
- **Viterbi**: The removal and preservation of nodes is considered as an optimization problem. For each leaf node, the path with the lowest cost to the root node is taken, where a cost is assigned to each transition. In this paper, we do not consider this rule. For details see Salembier et al. [14].
- **Direct**: A node $C_h^k$ is removed if $A(P_h^k) < \lambda$: its pixels are lowered in grey level to the highest ancestor which meets the criterion, its descendants are unaffected.
- **Subtractive**: As above, but the descendants are lowered by the same amount as $C_h^k$ itself.

Fig. 2(b) shows the results of filtering the signal in Fig. 2(a) using the subtractive decision rule.

3) Restitution: In a single pass through the destination volume, each voxel is assigned the new grey level of the corresponding node in the Max-Tree. It is this phase that we skip and replace by a visualization phase in this work.

B. Volume Rendering

Volume rendering is the process of generating 2-D images from 3-D data. It is pervasive in many application areas, such as medical imaging, biology, and computational fluid dynamics. Basically, two approaches exist: surface fitting methods and direct volume rendering methods [26].

Surface fitting methods extract constant-value surfaces representing the boundary between materials. By taking different threshold values, distinct isosurfaces can be extracted from the
volume data by fitting geometric primitives such as triangle
meshes to the boundaries that match the threshold [27].

Direct volume rendering methods [28] do not make use of in-
termediate graphical primitives, but make a direct mapping from
the volume data onto the view plane. A standard method, called
X-ray rendering, is to integrate the data values along the line of
sight. Maximum intensity projection (MIP) takes the maximum
value encountered along this line. This method is used often to
visualize magnetic resonance angiography (MRA) data. Most
other methods require a classification step, in which colors and
opacities are assigned to ranges of data values. The opacities are
used to make some parts transparent in order to reveal the inter-
esting parts.

Much effort has been spent on speeding up the fundamental
volume rendering techniques, see [29] for an overview. For iso-
surface visualization, an expensive step is to select the active
cells, i.e., those cells that intersect the isosurface. There are two
basic approaches: range-based search methods and seed set gen-
eration methods. The range-based methods use an interval \([a, b]\)
of the range of the scalar field values of a cell, where \(a\) is its min-
imum value and \(b\) its maximum. Given an isosurface threshold \(t\),
a cell is intersected by the isosurface if \(a \leq t \leq b\). A popular
range-based method is the interval tree [30], which is a balanced
binary search tree that stores the intervals, and which can be
queried efficiently. The seed set generation methods construct
a small group of cells, the seed set, which includes at least one
active cell for each possible isosurface. Starting from the seed
set, adjacent isosurface cells are found by propagation methods
[20], [31], [32].

Acceleration techniques for direct volume rendering often in-
volve ways to skip empty space, for example, by storing the data
in an octree [33]. There also exist transform domain approaches,
based on a wavelet decomposition of the data [34], [35] or a
Fourier representation [36], [37]. The best rendering speed is
currently provided by specialized hardware, such as VolumePro
[38], or texture mapping on standard PC graphics cards [17],
[39].

Practically all acceleration methods have one problem in
common: the underlying data structures cannot be updated
very easily if the volume data change as a result of, for instance,
filtering. In general, the data structure has to be completely
rebuilt, imposing large delays between filtering and rendering.
The standard hardware-based texture mapping approach is also
not directly usable, since the whole volume has to be transferred
from main memory to graphics card memory, a step which is
expensive. In the next section, we will introduce extensions
to the basic Max-Tree that will allow us to visualize the data
directly from the Max-Tree representation.

III. Attributes

Any attribute of a connected component can be used with the
Max-Tree. If it is required that the Max-Tree building phase is
short, it should be possible to compute the attribute increment-
ally. An example is the volume of a connected compo-
nent, which can easily be updated voxel by voxel. Other exam-
pies are the length of the diagonal of the bounding box or the
minimum enclosing box; the radius, diameter, or volume of the
smallest enclosing sphere; the moment of inertia; the perimeter,
or criteria derived from the perimeter, such as complexity, sim-
plicity, compactness, eccentricity, jaggedness, etc. [10], [14].
However, if the Max-Tree building process can be done off-line,
this restriction does not hold, and any shape or size attribute
can be used. Examples are radius of the largest inscribed circle,
skeleton length, etc.

Previously, Wilkinson and Westenberg [13] proposed a scale-
invariant attribute based on the moment-of-inertia tensor of each
object for vessel enhancement filtering. The moment-of-inertia
tensor \(I(C)\) of an object \(C\) can be defined as

\[
I(C) = \begin{pmatrix}
I_{xx}(C) & I_{xy}(C) & I_{xz}(C) \\
I_{xy}(C) & I_{yy}(C) & I_{yz}(C) \\
I_{xz}(C) & I_{yz}(C) & I_{zz}(C)
\end{pmatrix}
\]  

(2)

with

\[
I_{xx}(C) = \sum_C (x - \bar{x})^2 + V(C)/12 
\]

(3)

\[
I_{yy}(C) = \sum_C (y - \bar{y})^2 + V(C)/12 
\]

(4)

\[
I_{zz}(C) = \sum_C (z - \bar{z})^2 + V(C)/12 
\]

(5)

\[
I_{xy}(C) = \sum_C (x - \bar{x})(y - \bar{y}) 
\]

(6)

\[
I_{xz}(C) = \sum_C (x - \bar{x})(z - \bar{z}) 
\]

(7)

and

\[
I_{yx}(C) = \sum_C (y - \bar{y})(z - \bar{z}) 
\]

(8)
in which \(V(C)\) stands for the volume of \(C\), and \(\bar{x}, \bar{y}, \text{ and } \bar{z}\)
denote the centre of mass coordinates. The term \(V(C)/12\) in the
diagonal terms corrects for the moment-of-inertia contribu-
tion of each of the individual voxels, which are considered to
be small cubes. This correction is necessary for two reasons:
1) it ensures scale invariance more precisely, and 2) it prevents
division-by-zero errors caused by zero eigenvalues. Tensor \(I\)
is essentially the covariance matrix multiplied by the number
of voxels in \(C\). It can be computed for each node of the tree by
computing \(\sum x^2, \sum y^2, \sum z^2, \sum xy, \sum xz, \sum yz, \sum x, \sum y,
\sum z\), and volume. Each of these is easily updated incrementally.

Several moment-invariants can be computed from the mo-
ment-of-inertia tensor. The noncompactness attribute \(N(C)\)
used for vessel enhancement [13] is defined as

\[
N(C) = \frac{\text{Tr}(I(C))}{V^{5/3}(C)}. 
\]

(9)

It can be shown that this is a 3-D counterpart of the first moment
invariant of Hu [40], and it was previously proposed outside the
filtering context in [41]. Other 3-D moment invariants are dis-
cussed in [42]. Here, we derive other moment-invariants which
have a straightforward geometric interpretation from the eigen-
values of \(I\). Let \(\lambda_1(C), \lambda_2(C), \text{ and } \lambda_3(C)\) be the three (real)
eigenvalues of \(I(C)\), sorted such that

\[
|\lambda_1(C)| \geq |\lambda_2(C)| \geq |\lambda_3(C)|. 
\]

(10)
Intuitively, $\lambda_1(C)$ is the variance along the major axis, multiplied by the volume of the object, and likewise for the other two eigenvalues. Therefore, the ratios

$$\mathcal{E}(C) = \left| \frac{\lambda_1(C)}{\lambda_2(C)} \right|$$

and

$$\mathcal{F}(C) = \left| \frac{\lambda_2(C)}{\lambda_3(C)} \right|$$

are measures of elongation and flatness, respectively. Furthermore, the lengths $d_i(C)$ of the principal axes can be estimated as

$$d_i(C) = \sqrt{20 |\lambda_i(C)| / V}$$

Finally, the sparseness $\mathcal{S}$ can be computed as

$$\mathcal{S}(C) = \frac{\pi d_1 d_2 d_3}{6 V}$$

i.e., it is the ratio of the volume expected given its principal axes, over the actual volume. It attains a minimum of 1 for solid spheres and ellipsoids, is equal to about 1.127 for solid rectangular blocks, and becomes higher as the object becomes either hollowed out or tree-like.

IV. AUGMENTED MAX-TREE

In the usual implementation of the filtering phase, the Max-Tree structure is modified by removing nodes from the tree. In order to be able to reuse the same Max-Tree for filtering with different values for the threshold parameter $\lambda$, nodes should not actually be removed from the tree. Therefore, we explicitily store the grey value of a node $C^k_h$ in an extra field. We introduce the notation $g(C^k_h)$ to denote this current grey value. The current grey value is initialized as $g(C^k_h) = h$, i.e., the original grey value of the node, and it is updated during filtering. The filtering phase now uses the original grey value of a node $C^k_h$, and stores the grey value resulting after filtering as $g(C^k_h)$.

We will replace the restitution phase, since reconstructing the complete volume is not convenient for visualization purposes. Instead, we introduce a few extensions that allow us to use the Max-Tree representation directly for visualization.

A. Splatting

Splatting is a visualization method that traverses the voxels and calculates the projection and contribution of a voxel to the pixels in the view plane [16]. In the method, the underlying continuous function is reconstructed from the discrete volume data by convolution with a reconstruction filter, followed by a mapping to the view plane by superposition of building blocks called footprints. The footprints are the result of integrating the reconstruction kernels along the line of sight. For orthographic projection, the footprints are the same for all voxels, and have to be computed only once for a given viewing direction.

MIP and X-ray rendering can be implemented very efficiently [35], [43], since the voxels can be processed in arbitrary order. Though it is possible to perform semi-transparent rendering by depth sorting the voxels, we restrict our splatting implementation to MIP and X-ray rendering, since splatting is to be used as a previewing method. A more sophisticated rendering algorithm is presented in Section IV-B.

Before we can describe the rendering algorithm, it is necessary to establish a mapping from Max-Tree nodes to voxels. During the construction phase, we fill a list $L$ with voxel coordinates in a lexicographical order according to grey value and Max-Tree node index. In each Max-Tree node, two extra fields are maintained: an offset in the list $L$, and the number of voxels that belong to that node. In essence, the list $L$ is just a flattened representation of the hierarchical queues, which are used in the flooding procedure.

The rendering algorithm starts at the root node, and traverses the tree to the leaves. At each node $C^k_h$, the current grey value $g(C^k_h)$ is inspected, and if $g(C^k_h) > 0$ (the background color), the voxels belonging to the node are looked up in the list $L$. For all voxels, we calculate the projection $p$ of the center of each voxel $v$ on the view plane. For MIP, $v$ only contributes to the pixel closest to $p$. For X-ray rendering, we distribute the contribution linearly to the four pixels closest to $p$. The final step is to close holes that appear for nonaxial views. For X-ray rendering, this is done by convolving the image with the splatting footprint [35]. Since linear interpolation does not combine very well with MIP [44], we use morphological sampling [45], and it has been shown that the footprints from classical splatting have to be replaced by a morphological closing in the view plane [43].

B. Hardware-Assisted Volume Rendering

Modern graphics cards support 2-D or 3-D texture mapping functionality, which can be used to implement volume rendering based on texture slicing [17], [39].

Volume rendering by 3-D texture mapping works as follows. The volume data are loaded as a 3-D texture $T$ in texture memory on the graphics hardware. Modern cards usually have 256-MB or 512-MB memory available. Next, polygons parallel to and at increasing distances from the view plane are generated. Resampling is then performed at the intersections of these polygons and $T$, a process during which trilinear interpolation is used. The resulting scalar values are then mapped to color and opacity by a transfer function that is also stored on the hardware as a 1-D texture. This is called a dependent texture lookup: the interpolated scalar value is used as an index in the transfer function texture. Finally, the polygons are blended in a back-to-front order to make the final image. The whole process takes place in hardware, and it can be implemented in a straightforward way in OpenGL and a so-called fragment program. A fragment is simply a pixel with associated color and depth value. In OpenGL, it is possible to replace the standard fragment operations, such as texture mapping, color operations, fog calculations, for example, by a fragment program that performs some other operation.

A bottleneck in the approach above is volume reconstruction from the Max-Tree and transfer of the volume data from main memory to the graphics card. For a moderately-sized data set of $256^3$ voxels, this can take in the order of 500 to 5000 ms, depending on the hardware and the size of the Max-Tree. This
is more than one order of magnitude slower than the Max-Tree filtering phase (see also Section V). We, therefore, propose a slightly different approach.

We construct a 1-D array \( A \) in main memory, containing all Max-Tree nodes with no specific ordering. Then, we make a volume data set \( T_v \) that for each voxel contains the index of the corresponding Max-Tree node in the array \( A \), rather than the node’s grey value. The volume \( T_v \) is transferred once to the graphics hardware. From the array \( A \), we derive a 1-D texture \( T \) that contains the current grey values of the Max-Tree nodes. This texture is updated and transferred to the graphics hardware each time filtering is applied (see \textit{Update} in Alg. 1). Transferring \( T \) will be much faster than transferring the whole volume data set, since in general the size of \( T \) is much smaller than the number of voxels. The rendering phase now requires one extra dependent texture lookup to obtain the grey value at a specific point when resampling \( T_v \). Sampling a voxel \( v \) of \( T_v \) yields an index, which is used to find the grey value of \( v \) in \( T \). A second dependent texture lookup occurs during the application of the transfer function, see \textit{Draw} in Alg. 1.

\begin{algorithm}
\caption{Pseudo code for texture-based volume rendering}
\begin{algorithmic}
\Function{Update}{ }
\State // \( g(p) \) denotes current grey value of node \( p \)
\For{\( i \leftarrow 0 \) to \( \lvert A \rvert \) do}
\State \( T[i] \leftarrow g(A[i]) \)
\EndFor
\State Transfer \( T \) to graphics hardware
\EndFunction
\Function{Draw}{ }
\For{all slice planes \( s \) do}
\For{all fragments \( f \) in \( s \) do}
\State \( i \leftarrow \) sample \( T_v \) in point \( f \) // Fetch node index
\State \( g \leftarrow T[i] \) // Fetch current grey value
\State \( f \text{color} \rightarrow \text{TransferFunction}(g) \)
\EndFor
\State Blend \( s \) with frame buffer
\EndFor
\EndFunction
\end{algorithmic}
\end{algorithm}

Due to limitations of the hardware, the actual implementation is somewhat more involved than sketched above. The first restriction is that textures can only have dimensions that are powers of two. Secondly, it is not possible to store integer-valued volume data directly, since the hardware only supports RGBA (red, green, blue, and alpha) components with eight bits per component. Furthermore, the size of a 1-D texture is usually also limited to about 4096. Therefore, we factorize the integer indices into RGB triplets: \( T_v \) becomes a 3-D RGB texture and \( T \) is turned into a 3-D texture. For example, consider a Max-Tree that contains 38868 nodes. Then, \( T \) will have dimensions \( 256 \times 256 \times 1 \), and the node with integer index 2698 is factorized as \((B, G, B) = (10, 138, 0)\). This way of storing Max-Tree node indices limits the maximum number of nodes to \( 256^3 \). We cannot use the alpha component, since 4-D textures are not supported.

During resampling, we have to take care that \( T_v \) contains indices, so that trilinear interpolation cannot be used directly. This problem is solved by a fragment program which performs the interpolation. The input to the fragment program is the 3-D coordinate of the sample point \( p \) at which we need the grey value. The first step is to determine the coordinates of the eight voxels in \( T_v \), closest to \( p \) by simple rounding operations. This yields the indices to be used to access \( T \) to obtain the current grey values of these voxels. These grey values are then trilinearly interpolated to obtain the grey value at \( p \). Finally, the transfer function is applied.

Since trilinear interpolation requires eight dependent texture lookups, it is computationally expensive. Therefore, we have also implemented a fragment program that performs nearest neighbor interpolation, i.e., it just finds the grey value of the voxel \( v \) closest to \( p \). This requires only one dependent texture lookup, at the cost of reduced image quality.

\subsection{C. Isosurfacing}

To extract an isosurface efficiently from the Max-Tree representation, we need a few more extensions, which we introduce below. First, consider the following observations.

Let a cell \( c \) be identified by the corner voxel with the minimum coordinate values within the cell. Define the root path of a node \( C_k^h \) as the path through the tree that starts in \( C_k^h \) and ends in the root node, and which contains all nodes encountered on the descent. In case of a 26-connected neighborhood, two important properties hold: 1) all eight corner voxels of any given cell \( c \) are part of the same root path in the Max-Tree, since they are all neighbors of each other; 2) filtering in the Max-Tree does not change the grey-level order of Max-Tree nodes along a root path. To see that the first property holds, consider the following cases for two arbitrary voxels \( p \) and \( q \) of a cell \( c \). If they have the same grey value, i.e., \( f(p) = f(q) \), they belong to the same flat zone and, thus, the same Max-Tree node \( C_k^{f(p)} \). If \( f(p) < f(q) \), both voxels belong to the same connected component in the threshold set \( X_{f(p)} \) and \( q \) is part of a flat zone of a child node of the Max-Tree node \( C_k^{f(q)} \). In case \( f(p) > f(q) \), the parent-child relation of \( p \) and \( q \) is inverted. The fact that the second property holds follows directly from the filtering rules discussed in Section II. Because of these properties, just one Max-Tree node \( C_k^h \) defines a cell’s minimum value and just one other Max-Tree node \( C_k^{f(q)} \) its maximum value. It is important to realize that, after filtering, the same nodes still define the cell’s minimum and maximum value, even though the actual grey values of the nodes themselves may have changed.

We construct two label volumes, \( V_{\text{min}} \) and \( V_{\text{max}} \), which store for each cell \( c \) the Max-Tree node containing its minimal and maximal neighboring node, respectively. Note that, if the original volume has size \( N_1 \times N_2 \times N_3 \), there are \((N_1 - 1) \times (N_2 - 1) \times (N_3 - 1) \) cells. Next, we define edge cells as those cells \( c \) for which \( V_{\text{min}}(c) \neq V_{\text{max}}(c) \). These are the cells that possibly contain an isosurface.
The idea now is to store each edge cell \( c \) in the Max-tree node \( V_{\text{max}}(c) \). We store these edge cells in ascending order of the current grey value of their minimum neighboring Max-Tree node. Initially, the current grey level equals the original grey level \( g(V_{\text{min}}(c)) \). Because all lower neighbors of a Max-tree node must belong to the same root path, and because the filtering retains the order of nodes in a root path, we may sort the edge cells assigned to each node in ascending order of their grey value. Fig. 3 shows an example of the augmented Max-Tree for a simple 4 × 4 image.

The memory overhead of edge cell administration is \( O(M + N) \), where \( M \) denotes the number of Max-Tree nodes and \( N \) the total number of cells. Assuming that \( s \) denotes the number of bytes required to store an integer or a pointer to a Max-Tree node, the exact memory overhead is \( s(M + 2N) \) bytes. \( V_{\text{min}} \) contains \( N \) pointers to Max-Tree nodes, and each Max-Tree node contains an array of indices of edge cells and the size of this array. Note that \( M \) is usually much smaller than the number of voxels, but these numbers can become equal in the worst case.

Given an isolevel query value \( t \), it is now straightforward to select the cells for which maximum and minimum grey values straddle this threshold (active cells). Because the leaves of the Max-Tree contain regional maxima, it is possible to descend from each leaf for which the current grey level is higher than \( t \). At each node \( p \) that is visited, the algorithm checks whether it has not yet been processed and whether the current grey level \( g(p) \geq t \). If so, the edge cells \( e_i^p \) of \( p \) are processed in ascending order, while the minimum filtered grey level of its minimum \( f(V_{\text{min}}(e_i^p)) \leq t \). When ready, the node is marked as processed, and the algorithm descends to the parent, until the root node is reached. The pseudo code for active cell selection is shown in Algorithm 2.

![Algorithm 2 Pseudo code for active cell selection](image)

The final step is to process the active cell list, and perform triangulation within each cell. This is done by analyzing the marching cubes cases [27], and constructing a triangulation accordingly. The normals of the corner voxels of the cell are re-computed each time the isolevel \( t \) or filter threshold \( \lambda \) changes. The triangulated isosurface is sent to the graphics hardware by means of OpenGL vertex arrays.

V. RESULTS

To measure the performance of our method, we used two data sets available publicly from http://www.volvis.org and a third one from http://www9.informatik.uni-erlangen.de/Ex-ternal/vollib. The data sets are foot (courtesy Philips Research, Hamburg, Germany), a rotational b-plane X-ray scan; aneurism, a CT scan (courtesy Viaetronix, Inc., USA); and piggy bank, also a CT scan (courtesy M. Bauer, Computer Graphics Group, University of Erlangen, Germany). Visualizations of these data sets are shown in Fig. 4, and the data set details are listed in Table I, including the Max-Tree construction time and the number of Max-Tree nodes. All timings were performed on a PC with 8 GB of memory, two dual core 2.4-GHz Opteron 280 CPUs, and two NVIDIA GeForce 7900 GTX graphics cards equipped with 512-MB memory each. Only one CPU and one graphics card were used for the timings.

We measured the computation time of the Volume attribute and the attributes defined in Section III. Table II lists the results. We did not include the timing results of the flatness (12) and
Fig. 4. (a) Isosurface visualization of the piggy bank data set. (b) MIP rendering of the foot data set. (c) MIP rendering of the aneurism data set, filtered by a criterion based on the noncompactness attribute (thresholded at λ = 2.0).

**Table I**

<table>
<thead>
<tr>
<th>NAME</th>
<th>DATA SET SIZE</th>
<th>DYNAMIC RANGE</th>
<th>MAX-TREE NODES</th>
<th>M-T BUILD TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Piggy bank</td>
<td>11,796,480</td>
<td>12-bits</td>
<td>206,529</td>
<td>14.7</td>
</tr>
<tr>
<td>Foot</td>
<td>16,777,216</td>
<td>8-bits</td>
<td>279,513</td>
<td>16.4</td>
</tr>
<tr>
<td>Aneurism</td>
<td>66,846,720</td>
<td>8-bits</td>
<td>505,952</td>
<td>100.0</td>
</tr>
</tbody>
</table>

**Table II**

<table>
<thead>
<tr>
<th>DATA SET</th>
<th>VOLUME</th>
<th>NON-COMPACTNESS</th>
<th>SPARSENESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Piggy bank</td>
<td>109</td>
<td>226</td>
<td>383</td>
</tr>
<tr>
<td>Foot</td>
<td>151</td>
<td>329</td>
<td>538</td>
</tr>
<tr>
<td>Aneurism</td>
<td>457</td>
<td>1,047</td>
<td>1,796</td>
</tr>
</tbody>
</table>

The elongation (11) attributes, because they showed the same performance as the sparseness (14) attribute. This was expected, since the computation time in these three attributes is dominated by the eigenvalue calculation.

Fig. 5 contains timing results for browsing the filter threshold λ in the range [0, 25] for the elongation attribute (11), and subsequent direct volume rendering (MIP). All times are given in milliseconds. The plots compare splatting directly from the Max-Tree (Section IV-A) with our texture-based volume rendering approach (Section IV-B) for both nearest neighbor (MTNN) and trilinear interpolation (MTTL). We have also included a standard texture-based volume rendering approach (STBV). In all cases, the window size for displaying the final image was 400 × 400 pixels, and the number of slice planes for texture-based volume rendering was fixed to 300. Table III, in addition, lists the average times to perform filtering, transfers of data to the graphics hardware, and drawing time for the texture-based volume rendering methods. From the table, it is clear that filtering can be done at interactive rates. We must note here that the filtering time depends only on the number of Max-Tree nodes, and that it is independent of the choice of the filter criterion, which only affects the number of voxels in the filtered data set. Splatting and STBV are both highly dependent on the number of voxels that remain after filtering. Our approaches MTNN and MTTL show an almost constant behavior over the whole λ range. We can see that volume reconstruction and transfer in STBV is very costly, and that our method performs much better, because much less data has to be transferred.

In Table IV, we show the average frame rates (the number of times per second all data can be drawn to the screen) of all direct volume rendering methods. These rates were measured twice with over a full rotation around the z axis in increments of one degree. The first timing was performed with λ = 0 to measure performance on the raw data sets. For the second timing, we used a different criterion for each data set: for foot, the elongation attribute worked well on the bones, the flatness attribute revealed the coins in piggy bank, and the noncompactness attribute enhanced the vessels in aneurism. The threshold λ was set to a value that yielded a first interesting result, i.e., the moment components that do not satisfy the filtering criterion start to disappear. Splatting works only well for very low voxel counts. As expected, STBV is the fastest, because it requires only one dependent texture lookup during drawing. Our approach also shows high frame rates for nearest neighbor interpolation, and it is still quite fast for trilinear interpolation. During a rotation, or other user interaction such as zooming and panning, we, therefore, use nearest neighbor interpolation, and switch to trilinear interpolation when interaction ceases. Nearest neighbor interpolation has lower image quality (see Fig. 6), but it is acceptable during user interaction.

We also compared the performances of our isosurface extraction method and the interval tree. In the experiment, we increased the isosurface threshold t from the lowest data value to the maximum data value in steps of one or ten, depending on the dynamic range of the data. The performance measurement was carried out on the filtered data sets listed in Table IV. The interval tree had an overall somewhat better performance, which is not surprising, since it is optimized for the purpose of isosurface browsing. However, if we take interval tree construction time...
Table III

Average times (in milliseconds) for browsing the filter threshold \( \lambda \) in the range [0, 25]. The second column lists the filtering time. The third column lists the time to transfer the lookup table \( T \) to the graphics hardware, and the fourth and fifth column show the drawing times for MTNN and MTTL, respectively. The last two columns list the time to reconstruct and transfer the filtered volume data to the graphics hardware and the drawing time for STBV, respectively.

<table>
<thead>
<tr>
<th>Data set</th>
<th>Filter</th>
<th>Max-Tree</th>
<th>STBV</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>MTNN</td>
<td>MTTL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>transfer</td>
<td>draw</td>
</tr>
<tr>
<td>Piggy bank</td>
<td>20</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>Foot</td>
<td>27</td>
<td>51</td>
<td>19</td>
</tr>
<tr>
<td>Aneurism</td>
<td>33</td>
<td>92</td>
<td>276</td>
</tr>
</tbody>
</table>

Table IV

Average frame rates measured over a full rotation around the \( z \)-axis. For each data set, another attribute was used (see text for details). We filtered at \( \lambda = 0 \) (raw data) and \( \lambda \) set to a value for which the first interesting result was obtained.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>MTNN</th>
<th>MTTL</th>
<th>STBV</th>
<th>Splatting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Piggy bank</td>
<td>( \lambda = 0 )</td>
<td>54.5</td>
<td>7.2</td>
<td>86.1</td>
</tr>
<tr>
<td>Foot</td>
<td>( \lambda = 0 )</td>
<td>44.4</td>
<td>7.1</td>
<td>91.4</td>
</tr>
<tr>
<td>Aneurism</td>
<td>( \lambda = 0 )</td>
<td>4.1</td>
<td>0.5</td>
<td>63.4</td>
</tr>
<tr>
<td>Piggy bank</td>
<td>( \lambda_{E} = 2.0 )</td>
<td>56.5</td>
<td>7.2</td>
<td>94.2</td>
</tr>
<tr>
<td>Foot</td>
<td>( \lambda_{E} = 1.5 )</td>
<td>57.4</td>
<td>7.3</td>
<td>93.0</td>
</tr>
<tr>
<td>Aneurism</td>
<td>( \lambda_{NC} = 2.0 )</td>
<td>3.9</td>
<td>0.5</td>
<td>67.5</td>
</tr>
</tbody>
</table>

Fig. 5. Timing (ms) for browsing \( \lambda \) in the range [0, 25] for all datasets. Elongation (11) was used as a filter criterion. (a) Piggy bank; (b) foot; (c) aneurism.

Fig. 6. Comparison between (left) nearest neighbor interpolation and (right) trilinear interpolation.

into account, which ranges between 0.9 and 56.0 s for our data sets, our approach that extracts an isosurface directly from the Max-Tree is more suitable when a user wants to perform both isovalue browsing and \( \lambda \) browsing.

Fig. 7 illustrates the effect of applying various attributes to the piggy bank data set. We used isosurfaces to visualize the results. The coins inside the piggy bank, which, incidentally, are made of chocolate in order to prevent scattering artifacts from the CT scanner, can be separated well by using the flatness attribute and a high threshold (\( \lambda = 100.0 \)), see Fig. 7(a). The piggy bank is standing on a wooden block, see Fig. 7(b). The wood structure is well preserved by the sparseness attribute and a threshold of \( \lambda = 6.5 \). There is a hole in the piggy bank’s belly, covered by a lid containing two springs that keep it in place. By filtering away nonelongated structures, these springs can be separated out (see Fig. 7). The elongation attribute was used with a threshold of \( \lambda = 5.0 \). Since our approach enables changing of attributes and thresholds at interactive rates, the images in Fig. 7 could be made within a few minutes. It would have been difficult and time consuming to achieve the same results without the visualization methods proposed in this paper.

We would like to refer the reader to our web page (http://www.rug.nl/informatica/onderzoek/programmas/svcg/demos), which contains a number of movies and a demo program. These demonstrate the interactivity offered by our method well.

VI. Discussion

In this paper, we have proposed methods for interactive filtering and visualization of volume data sets based on a class of shape preserving filters. We have briefly reviewed such filters and explained how they can be implemented efficiently using Max-Trees, and we have also introduced new attributes based on 3-D moment invariants similar to those in [41] and [42]. The Max-Tree approach splits the filtering task in three stages. The first stage is a preprocessing step that involves the construction of a tree, while the second stage performs actual filtering using this tree. The third stage is reconstruction, which we have replaced by a visualization step. Building the tree takes several seconds for small volumes, and in the order of minutes for larger volumes.

We have proposed several extensions to the Max-Tree data structure that allow efficient visualization. Direct volume rendering based on splatting has turned out to be a viable approach when the data are sparse. For more dense data, a better performance is obtained by performing the rendering on graphics hardware. For this purpose, we have introduced a method to represent the Max-Tree as a lookup table on the graphics hardware. Then, filtering only requires an update of this lookup table, which is much faster than a traditional approach of volume reconstruction and subsequent transfer to the graphics hardware. We have also introduced extensions that
allow fast isolevel queries in the Max-Tree data structure. The performance comes close to the performance of the interval tree; however, our method has the advantage that both filter threshold browsing and isolevel browsing are fast. The interval tree cannot be used for both these purposes, since it requires a large amount of preprocessing time to construct the tree.

In the current comparison, the MTNN and MTTL methods come out best, but their drawback is that they require the entire label volume $T_v$ to be stored on the graphics card. This means that very large volumes cannot be visualized in this way. Both splatting and isosurfacing are feasible if graphics memory is insufficient, though splatting only works at adequate speeds if the volume is sparse, and isosurfacing is also only possible at interactive rates if the active cell count is low. For angiographic datasets, sparse volumes are obtained by filter threshold settings near the optimum.

In principle, both MTNN and MTTL could be extended to other connected filters, such as the autodual filters of Monasse and Guichard [23], or the even more general method of Soille [46]. This is because any connected filter only ever merges flat zones, or assigns new grey levels to them [47]. Thus, any connected filter can be constructed by first labelling all flat zones in a volume, computing a lookup table containing new grey levels for every flat zone, and computing the new volume by simple table lookup. If recomputing the lookup table can be done efficiently, it is possible to use a variant of MTNN or MTTL for rapid visualization.

The attributes stored for each voxel in the Max-Tree could be used in many different ways other than filtering. In particular, modifying color and opacity would be interesting ways to manipulate the visual representation of these volumes. As the Max-Tree provides a decomposition in connected components, a classification scheme could assign each of the components a different color, texture, or annotation. The speed of recomputation of these attributes, even if fairly complicated moment invariants are used, means such interaction can readily include changing the attributes themselves, rather than just changing visualization parameters such as transfer-function settings or iso-surface levels.

It might also be possible to perform progressive refinement based on the component structure stored in the Max-Tree, rendering the most important components first, and gradually adding less important ones. Unlike other progressive refinement methods, the shape preserving nature of connected filters would mean that the rendered components would appear at full resolution, rather than as a blurred version first [35], [43].

REFERENCES
