EG UK Theory and Practice of Computer Graphics (2007)
Ik Soo Lim, David Duce (Editors)

Speeding Up Isosurfacing: The *Matryoshka* Algorithm

Sérgio Lopes, Adriano Lopes and M. Próspero dos Santos

New University of Lisbon

Abstract

We propose a hybrid algorithm for isosurface visualization that embraces both polygon rendering and direct surface rendering concepts. It uses raytracing to achieve high image quality but avoids the associated empty ray traversal. At the heart of the algorithm is a caching strategy resembling the famous Russian stacking dolls, that allows the processing of cells of interest from any viewing orientation. We use an optimised interval tree to extract these cells from the volume. In that respect, we propose two versions, combining grouping of cells and ordering. In comparison to the classic version of the interval tree, the memory overhead decreases but the increasing in the query time is marginal.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation. I.3.6 [Computer Graphics]: Methodology and Techniques. I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism.

1. Introduction

Volume visualization aims to generate meaningful and accurate images from volumetric scalar data. The main techniques are *volume rendering* and *isosurfacing*. While volume rendering tries to visualize the volume as a whole, isosurfacing looks at the visualization of particular isosurfaces within the volume.

Within isosurfacing, there are two approaches for generating images: *polygon rendering* and *direct surface rendering*. In both cases, the main goal is to process/visualize the *active cells*, which are the cells in the volume that intersect a given isosurface. In general, polygon rendering methods nowadays divide the visualization process into two steps: a preparation step, usually named as *isosurface extraction*, where active cells are located within the volume, and a second step where a triangle mesh approximation is generated and subsequently visualized. The main strength of these methods relies on the use of graphical hardware to visualize the mesh. Although no triangles are generated for cells that do not contribute to the isosurface, thousands or even millions of triangles can be generated, some of them very tiny, flooding current graphical hardware below interactive rates in these cases. Direct surface rendering methods take a unified approach as the objective is to use raytracing for a direct visualization of the isosurfaces. The basic idea is to have rays traversing the volume and computing the intersections that are found. As such, the visualization process also depends on cells that do not contribute to the final image due to ray traversal.

In this paper, we present a pure software algorithm for isosurfacing, built up on the idea of joining strengths from both direct surface rendering and polygon rendering methods.

The algorithm separates isosurface extraction from its visualization. As the rendering bottleneck for ray tracing is empty space traversal, which is similar to the problem of inactive cells in polygon rendering methods, we can adopt existing polygon rendering acceleration techniques. Hence, the isosurface extraction uses optimised versions of an interval tree data structure to reduce memory footprint. Then the visualization follows local raytracing to produce high-quality isosurface rendering.

The visualization is supported by a new view-dependent caching strategy that allows rapid updates on rotation of views by treating the dataset as a set of shells from the outside in. Indeed, a typical isosurface interaction has an ex-

© The Eurographics Association 2007.

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

traction stage followed by a manipulation phase, and in the latter, there is no need to re-extract the active cells.

2. Related Work

Isosurfacing was first tackled with polygon rendering methods such as the Marching Cubes algorithm [LC87]. These methods start by extracting a polygon mesh from the volume, representing an approximation to the chosen isosurface, and then the polygons are directly rendered using graphical hardware.

Initially, there was a look-up table for the Marching Cubes algorithm, establishing a correspondence between cell data values and triangulation. But some inaccuracies were found in the process and successive improvements were proposed to fix the algorithm, for instance, at the level of face and internal cells ambiguities, as well as accuracy [NH91, Nat94, LB03].

Isosurface extraction developed itself as a research area. A naive algorithm would visit every cell in the volume in order to verify if the cell is intersected by an isosurface, yielding to a computational time complexity of $O(n)$, with n being the total number of cells of the volume. So research efforts were pursued in order to lower this time complexity.

In 1996, Livnat et al. [LSJ96] introduced the idea of the span space, a 2D space where each cell is represented by a 2D point (x, y). The x component is the minimum value of the cell and the y component is the maximum of the cell. Since the relation $minimum \leq maximum$ holds, all the cells can be represented in the line equation $y = x$ or in the subspace above it. Using the span space, the problem of finding the active cells that intersect an isosurface of isovalue q is equivalent to finding all the points that are within the rectangle defined by $y \geq q$ and $x \leq q$. In the original research, Livnat used a kd-tree to do so. This finding method had a complexity of $O(\sqrt{n} + k)$, where k is the number of active cells. Later, Livnat updated the algorithm with a lattice subdivision scheme which has led to a slightly better performance [SHLJ96].

Besides the kd-tree, the interval tree is another data structure suitable to answer query range of data values [BvKOS00, Ede80]. Cignoni et al. [CMPS96] were the first to join the concepts of span-space and interval trees for active cell location. The interval tree works by processing each cell as an interval on the real line, being the interval defined by two values: the cell minimum and maximum values. The answer to which cells intersect a given isosurface is answered with a time complexity of $O(log(n) + k)$.

Notice that the rendering time is directly proportional to the polygons that represent the isosurface, which, by themselves, only depend on the active cells. Therefore, cells that do not intersect the isosurface are not even considered at the visualization stage. However, in the recent years, the size of datasets has been growing. Even if there are few polygons per cell, the number of polygons can reach the order of millions quite easily. Despite the recent advances in computer graphics hardware, this amount of polygons can overflow the capacity of current hardware. Moreover, some polygons can be smaller than a single pixel after being projected into the image plane. This implies that the computational effort due to such polygons is huge considering the small part of the image they generate.

Direct Surface Rendering has received great attention in the recent past. The use of raytracing for isosurface rendering was first approached by the seminar work of Parker et al. [PSL*98, PPL*99], where a distributed shared-memory multiprocessor machine was used to allow interactive frame rates. Their work showed that raytracing is a practical alternative to polygon rendering, especially in the case of large datasets. Their algorithm consisted of three steps:

1. Traversing a ray through cells which do not contain any isosurface.
2. Computing analytically the isosurface when intersecting a voxel that contains the isosurface.
3. Shading the resulting intersection point.

Additionally, optimisations at the level of ray traversal and at caching behaviour using memory bricking were quite relevant to achieve interactive frame rates [PSL*98] . The same authors reported that the bottleneck was in the first step of the algorithm. As an example, for the Visible Woman dataset [Erl], ray traversal was accountable for 55% of the time while visualizing a skin isosurface, and 66% of the time while visualizing a bone isosurface.

Direct isosurfacing then became available in common desktop computers. It is the case of the work of Neubauer et al. [NMHW02] using an Intel Pentium 4 1900 MHz single processor. Their method was particularly designed for practical applications of post-implantation assessment of endovascular stent placement. Optimisations on their work included a faster empty space traversal, data reduction and improvement of caching behaviour using a conjunction of minmax octrees and macro-cells. But they reported only 25% of the frame time for initialising and tracking local rays, as opposed to the ray traversal in Parker's work. We note, however, that the rays in Neubauer's work stopped at the first hit reached so there was no opacity level associated to isosurfaces.

In 2005, Wald et al. [WHFS05] used an implicit kd-tree for isosurface raytracing. Besides the use of the kd-tree to speed up ray traversal, they also used SIMD instructions for the same ray traversal step and for ray-isosurface intersection calculations. As Parker and colleagues did, they also reported the dominance of ray-traversal steps. Indeed, they recognised that traversal steps were much more common than ray-isosurface intersections, and usually dominated over the time spent on intersections. For example, in a

© The Eurographics Association 2007.

reported test, about 67% was spent on ray traversal, as opposed to only 9% on intersections.

All in all, the bottleneck of isosurfacing using raytracing lies in the empty ray traversal phase. We should note that this observation can be a little counter intuitive, as one could wrongly assume that, when using raytracing in this context, the bottleneck would be in the ray-isosurface intersection calculation. We recall that, in this case, traversal steps are much more common than intersection calculation steps, as acknowledged by Wald et al. [WHFS05].

3. Overview of the *Matryoshka* Algorithm

The isosurfacing algorithm we propose, which envisages the use of concepts from both polygon rendering and direct surface rendering, consists of three major stages, as depicted in Figure 1: isosurface extraction, isosurface visualization and an in-between storage of extracted active cells.

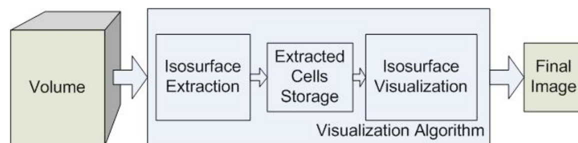

Figure 1: *The different phases of the isosurfacing algorithm.*

We are considering structured data volumes and parallel projection. Isosurface extraction, described in Section 4, relies on the use of interval trees. We will see in Section 5 how these cells are stored and how they are retrieved in a front-to-back order, according to the viewing camera. Notice that isosurface extraction is only done when the user changes the threshold values of isosurfaces. Each of the active cells obtained in a front-to-back order is processed according to the pipeline depicted in Figure 2. We now describe the details of each phase.

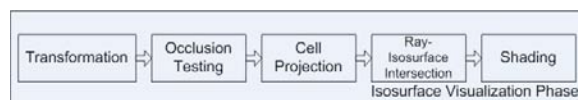

Figure 2: *Isosurface visualization pipeline.*

Transformation. This phase is responsible for obtaining the corresponding world and camera coordinates. The world coordinates are obtained by applying the volume's scales to the cell's coordinates. The camera coordinates correspond to a set of 2D points representing the projection of the 3D cell in the image plane. Since the original cubic cell has a convex shape, the resulting projection is a 2D convex polygon having either four or six vertices. In practice, if we consider all the eight points projected in the image plane, the resulting polygon is the convex hull of this set of points. Since we are using a parallel projection, the convex hull has the same shape for every cell. On that basis, we only need to compute the convex hull for the first cell that is being processed in a frame and apply this result to the other cells in the frame.

Occlusion testing. A cell is tested in camera coordinates with the opacity buffer. If all the pixels are already full (a pixel value in the opacity buffer close to 100%) then the cell is rejected and the pipeline restarts with another cell. However, finding the exact pixels of the polygon that are projected to camera coordinates involves a complete rasterization of the polygon. In order to make this phase faster, only the pixels between the cells camera minimum and maximum coordinates in x and y are tested. This may lead to unnecessary pixel verifications in some cases, but there is no overhead in projecting the polygon. Also, if it is detected that a cell does not project to a pixel, it is rejected in this phase. This particular case can happen when a cell is smaller than one pixel and ends up being projected in-between pixels.

Cell projection. Here, the polygon representing the cell in screen coordinates is rasterized. For each of these individual pixels, we test again in the opacity buffer. The purpose now is to eliminate from consideration pixels that are already full.

Ray-Isosurface intersection. For the pixels that are not completely full, we compute the intersections between a ray and the isosurface inside the cubic cell. In general, this computation requires solving a polynomial of degree 3. To do so we use the algorithm as shown by Schwarze in [Sch90]. Other faster algorithms may exist [MKW*04] but their solutions may not be entirely accurate. The intersections inside the cell are sorted in a front-to-back order according to the ray direction. For each of these successful intersections, we perform shading calculations to add realism to the final image. For the shading, we estimate the normals from the volume using central differences. The final colour also depends on the isosurface color and opacity. Following the front-to-back reasoning, if the opacity level of a certain pixel reaches 100%, then no more intersections are performed for that pixel.

4. Isosurface Extraction

In order to obtain the active cells from the volume, we have opted for an interval tree data structure. This decision is justified by its optimal complexity for such query [CMPS96]. As mentioned in Section 2, the interval tree obtains the active cells for one particular isosurface with a time complexity of $O(log(n)+k)$, so it is time-efficient. We should stress that, in the case of scalar data volumes, the term k completely dominates over $log(n)$ as the number of active cells is usually of the order of hundreds of thousands or even millions.

Nevertheless, an interval tree is also considered as space-inefficient. In order to overcome such drawback, one should

© The Eurographics Association 2007.

think about an hybrid method that uses larger blocks with the individual cells stored in a sorted order.

4.1. Spatial Complexity of Interval Trees

The properties of the interval tree data structure [BvKOS00, Ede80] state that memory requirements for storage are of the order of $O(n)$. Assuming that the volume's dimensions are given by Δx, Δy and Δz, then there is a total of $(\Delta x - 1)(\Delta y - 1)(\Delta z - 1)$ cells. In the case of eight bit volumes, the volume's memory overhead is about $(\Delta x \Delta y \Delta z)$ bytes. The interval tree holds pointers to the cells in two auxiliary lists, being each cell referenced by two pointers. Considering a pointer size of four bytes, the space overhead is of $8(\Delta x - 1)(\Delta y - 1)(\Delta z - 1)$ bytes. If on top of that we add the loading of the volume, then the total space usage is of nine times the size of the volume. This requirement can be prohibitive in some cases.

4.2. Interval Trees with Cell Grouping

One way of decreasing the memory overhead of the interval tree is to have a pointer in the two lists representing more than one cell. The question here is how many and which cells a pointer would represent. The larger the number of cells a pointer represents, the smaller the interval tree is. However, care must be taken since relevant changes in the query procedure may increase the query time.

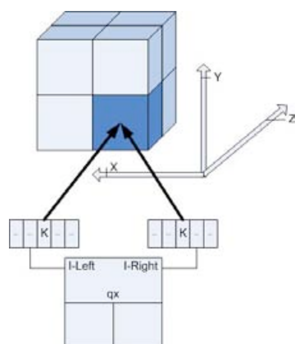

Figure 3: *A node of an interval tree with cell grouping.*

The first approach we have set is to have a pointer to represent a group of eight adjacent cells. By doing so, the total size for the entire interval tree would be about 1/8th of the size of a regular interval tree. Intuitively, it is as if we group a set of eight neighbour cells into a new bigger cell, and the interval tree stores these new big cells.

Notice that the groups of eight cells are disjoint. Figure 3 illustrates a node in this new tree: it is basically the same as the nodes in a normal interval tree, consisting in a value q_x and the two lists $I - Left$ and $I - Right$. But each pointer now references a set of eight adjacent cells, and, inside the group of eight cells, the one with the lowest coordinates is taken as the representative of the set.

The changes in the construction of the tree are minimal, requiring only that, when the minimum and maximum of a cell are obtained, they are actually the global minimum and maximum of the eight cells as a group. Besides the reduction of the size of the lists to 1/8th of the regular interval trees' size, the height of the new tree is also smaller since we are dealing, for practical purposes, with 1/8 of the number of cells.

The implementation of the query process is simple, because, while we are visiting one of the lists, we can stop iterating as soon as we find a group of eight cells that does not intersect the query value q. Notice that, for each of these bigger cells, we still have to check each of the eight composing cells.

The original tree has a query complexity of $O(log(n) + k)$, with k being the dominating factor. For this new tree, the worst case scenario occurs when an isosurface intersects 1/8 of all the cells in the volume in such a way that it only intersects one cell from each group of eight cells. The term k then increases by a factor of eight since we must verify all the eight sub-cells, yielding to a complexity of $O(log(n) + 8k)$. Fortunately, this situation is not frequent with practical data volumes. The complexity of the query process is then directly dependent on the factor associated to the k term. It is rather difficult to estimate the exact value, as it not only depends on the volume but also on the specific isovalue we are querying. Section 4.4 provides test results from this new tree applied on several volumes.

4.3. Interval Trees with both Cell Grouping and Ordering

With the solution presented above, we can reduce the memory requirements for a general interval tree by a factor of 1/8. However, the complexity of the query process can become higher as we may have to check more cells. On that basis, we propose another interval tree, that lies somewhere in-between the original interval tree and the previous tree, both in terms of space and query time.

The basic idea is to keep the previous tree but somehow to speed up the query process. Based on the previous tree, each pointer to a group of eight cells is stored in two lists: one is sorted in ascending order by the minimum value of the cell, and the other one in decreasing order by the maximum value. If, for each pointer, we have the eight cells sorted by their minimum or maximum value (depending on the list where the pointer is stored) then we can reduce the number of cells that are actually tested. This is the main idea: to maintain the grouping of eight cells while storing ordering information for each of these cells. By doing so, the number of visited cells is lowered in comparison to the previous tree. However, it is not so easy to give an exact number of the extra cells

© The Eurographics Association 2007.

that are tested while comparing with the normal interval tree. This issue will be addressed in the next section, where the solutions are tested with different volumes.

The memory requirements of this new tree should be discussed as extra space is needed in order to keep each group of cells ordered. Here, for each group of eight cells, we use a vector of eight numbers indicating its order. Since each number is between one and eight, only three bits are needed. Hence, the total number of bits necessary to store the eight numbers is $24\,bits = 3\,bytes$. The total space overhead of this interval tree is then the same as in the previous one, but with three more bytes for each cell pointer. Again, assuming that the size of each pointer is four bytes, the space overhead for each cell is therefore $2(4+3)/8 = 1.75$ bytes.

4.4. Tests on Interval Trees

In order to evaluate the two proposed trees, tests were run on a PC platform with a Celeron 2000 processor and 512Mb of RAM.

Table 1 shows the results. We have chosen volumes with different sizes so tests are run under different conditions. All the volumes are eight bit volumes, publicly available at [vol]. The test consisted of querying the interval trees for all 256 isosurface values.

Figure 4 depicts the sizes of interval trees and respective query times. Both proposed trees are much smaller than the original interval tree. Notice that, depending on the available memory, certain volumes may not fit in memory. This is more noticeable in the case of a normal interval tree. For example, the size necessary for storing the Bonsai dataset in Table 1 is approximately 130Mb for the *Normal Tree*, as opposed to only 16Mb and 30Mb for the trees with grouping and with both grouping and sorting respectively.

In respect to query times, we conclude that the normal interval tree gives the lowest times, while the tree with only cells' grouping presents the slowest times. The approach using grouping of cells with sorting provides results that lie somewhere in-between the two other trees.

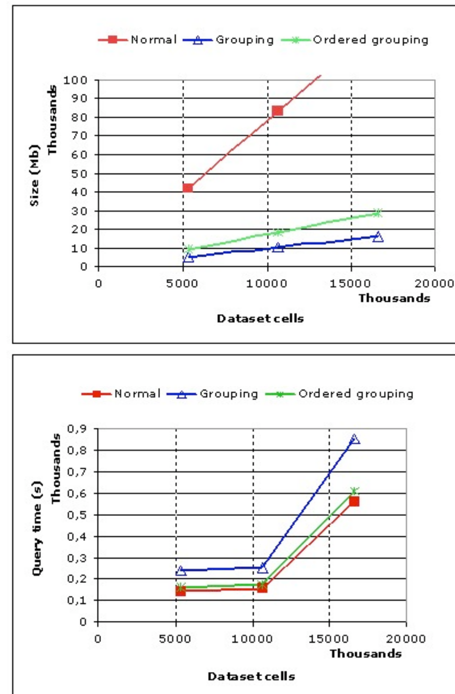

Figure 4: *Interval trees: size and query times, using the figures from Table 1.*

5. Isosurface Visualization

The interval tree allows an efficient query for the cells that intersect a given isosurface. Unfortunately, its query results are completely unordered since this data structure does not provide any kind of geometrical relationship among the returned cells. The consequences are particularly serious: if these unordered cells are rendered directly based on ray-tracing, we may end up with incorrect images. Indeed, the correct cell processing order in a front-to-back (or back-to-front) scenario depends on the viewing camera in 3D space.

We now present the *Matryoshka* data structure, that allows correct view-dependent visualization.

5.1. Cell Caching: the *Matryoshka* Structure

With the *Matryoshka* structure, we pass the resulting cells from isosurface extraction to the visualization phase in a form of layers within the volume. This approach slightly resembles the well-known principle behind the famous Russian stacking dolls (originally known as *Matryoshka*) where one doll completely encloses a smaller one and so on. Worth to mentioning at this point the work of Udupa and Odhner [UO93] from 1993, where shells of cells of interest were previously stored in lists, including extra information, and projected later on.

In our approach, a layer is composed of six planes that enclose a small volume. These planes are perpendicular to the three orthographic axes, and there are two for each axis. For practical purposes, each of these conceptual planes is simply a list of cells that belong to that plane and that also intersect the queried isosurface. Figure 5 illustrates the concept. If a plane does not contain any cells it will not even be taken into consideration. We note that a given layer completely encloses other layers and may itself be enclosed by a larger layer. Moreover, cells that can belong to both X and Y planes are stored in the X plane, cells that can belong to both X and Z planes are also stored in the X plane, and cells that can belong to Y and Z planes are stored in the Y plane.

If Δx, Δy and Δz are the dimensions of the volume in each

© The Eurographics Association 2007.

Table 1: *Test results for the interval trees. Both proposed trees are much smaller than a normal interval tree. The normal interval tree is the fastest one, the tree with just grouping is the slowest and the tree with both grouping and sorting delivers in-between times.*

DataSet	Resolution	Number of Cells	Interval Tree	Size (bytes)	Query Time (milliseconds)
Lobster	301x324x56	5,329,500	Normal	42,637,272	8,521.0
			Grouping of 8	5,443,797	14,221.3
			Ordered grouping of 8	9,526,197	9,613.7
StatueLeg	341x341x93	10.635.200	Normal	85,083,352	9,287.2
			Grouping of 8	10,635,902	15,240.1
			Ordered grouping of 8	18,612,302	10,567.3
Bonsai	256x256x256	16,581,375	Normal	132,653,067	33,607.5
			Grouping of 8	16,777,948	51,234.9
			Ordered grouping of 8	29,360,860	36,605.6

of the three main axes and m the smallest of these three values, then the total number of layers for the volume is given by $\lceil m/2 \rceil$. All the layers for a given isosurface can then be stored in a vector with size $\lceil m/2 \rceil$. The innermost layer, which is also the smallest one, may be a slightly different case from the rest of the layers: it can degenerate to be either a single plane, a single line or a single cell. Nevertheless, this particular case is dealt with as a general layer, in which most planes are non-existing.

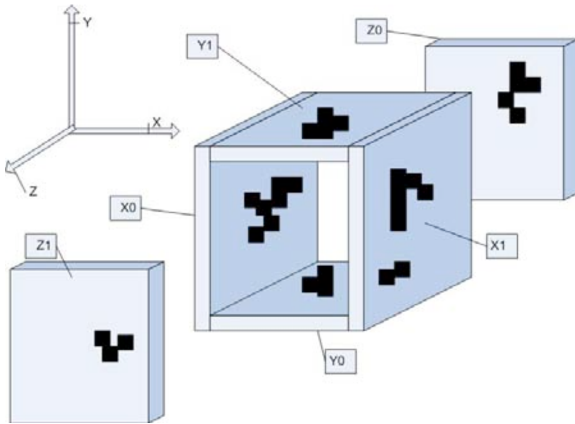

Figure 5: *In the* Matryoshka *data structure, a layer is composed of six connecting planes (expanded perspective in the picture). The black marks represent the active cells contained in the particular plane that are stored.*

The construction of the data structure is straightforward: for each active cell queried from the interval tree, we start by obtaining the layer to which the cell belongs to. This is achieved by looking at lower cell's coordinates, $cell_x$, $cell_y$ and $cell_z$. Let

- min_1 be the minimum of the values $cell_x$, $cell_y$ and $cell_z$;
- min_2 be the minimum of the values $(\Delta x - 1) - cell_x$, $(\Delta y - 1) - cell_y$ and $(\Delta z - 1) - cell_z$.

The index for the cell's layer is given by the minimum value of min_1 and min_2.

Once inside the layer, it is merely a question of inserting the cell into the appropriate plane, which again can be done by just considering the cell's coordinates:

- let x_0, x_1, y_0, y_1, z_0, z_1 be the values that define the six planes of the given layer, that is: $X = x_0$, $X = x_1$, $Y = y_0$, $Y = y_1$, $Z = z_0$, $Z = z_1$;
- the appropriate list is given by: if $cell_x = x_k$, with k equal to 0 or 1, the list is the one that represents the plane $X = x_k$. If not, the same thinking is applied for $cell_y$ and then for $cell_z$.

This insertion can be done in constant $O(1)$ time so the complexity of this construction is $O(k)$, with k being the number of cells. As for memory requirements, the overhead is also directly proportional to k, since the layers only contain a list of active cells that are stored once. Also, the insertion of a cell in the layers is merged into the query process of the interval tree. This allows the cells to be processed only once, so avoiding the need of a second round if the two processes were considered separately.

5.2. Active Cells' Visualization

In the visualization process, our goal is to process the cells in a front-to-back order along the direction of a parallel projection, which corresponds to the viewing direction of the camera. The question now is how to achieve that with the *Matryoshka*-like data structure.

Let us consider a general layer composed of six planes, as depicted in Figure 5, and let us analyse the relationship between cells contained in this layer and cells contained in the innermost layers. In a front-to-back traversal for all cells, the generic rendering process for the current cells' layer and for the next inner layers can be established as follows:

- Process the cells from this layer that possibly occlude cells from any inner layer.

© The Eurographics Association 2007.

- Process the inner layers.
- Process the cells from this layer that are possibly occluded from cells from any inner layer.

The missing detail here is how to classify the cells from a layer into occluding and occluded. Along a generic viewing direction, we can divide the six existing planes of the layer into two groups: closest to the camera position and furthest from it. The main idea is to process, first the cells from the planes closest to the camera, then the inner layers, and finally the cells from the layers furthest from the camera. In Figure 5, for example, $z = Z1$, $y = Y1$ and $x = X1$ represent the planes closest to the viewer position in the projection direction, whereas $z = Z0$, $y = Y0$ and $x = X0$ are the planes furthest from the viewer.

Moreover, there are some details to be taken into account concerning the planes of the same group. In particular, the special cells from the plane $x = X1$ that theoretically could also belong to the plane $y = Y1$ must be treated before any cells from the plane $y = Y1$. Also, the plane $z = Z1$ must be the last plane to be considered since some cells from the planes $x = X1$ and $y = Y1$ possibly occlude some cells from this plane. Inversely, some cells from the plane $z = Z0$ may occlude some cells from the plane $y = Y0$ that, by itself, may include some cells from the plane $x = X0$. So, the correct rendering sequence for this example would be to process the cells from the plane $x = X1$, $y = Y1$ and $z = Z1$, then to process the inner layers, and finally to process the cells from $z = Z0$, $y = Y0$ and $x = X0$.

In conclusion, the algorithm for a given layer l_m is as follows:

1. Process the cells from planes $X = X_{kx}$, $Y = Y_{ky}$, $Z = Z_{kz}$.
2. If l_m is not the last layer, process the next layer.
3. Process the cells from the planes $X = X_{kx'}$, $Y = Y_{ky'}$, $Z = Z_{kz'}$.

where kx, ky, kz are either 0 or 1, being kx', kx' and kx' the corresponding opposite value.

The viewing camera determines the values for kx, ky, kz, as they define the closest planes to the camera position. Also, when processing each plane, we use an auxiliary buffer. This is necessary because all the cells in each plane need to be processed in a front-to-back order. Because of this, two passages are made: one for marking the cells as "drawable" in the buffer, and another one to process the buffer in the wanted front-to-back order.

5.2.1. Visualizing Several Isosurfaces

The visualization of more than one isosurface at the same time, each of them with a particular opacity value associated, requires one *Matryoshka* layer for each isosurface. The visualization procedure for each layer is the same as described in the previous section. But now all the different isosurface layers are processed concurrently. In particular, for each layer, the same corresponding planes for each isosurface are processed according to the order described in the previous section.

The idea is to use the auxiliary buffer in a more generic way. Given a set of the same planes corresponding to different isosurfaces, all the active cells are marked as drawable in the auxiliary buffer. Then this buffer is processed following an order that depends on the viewing camera. With this, the cells are correctly processed in a front-to-back order and with the correct ordering in relation to the cells on the same layer.

5.3. Tests and Results

The tests were run on an Intel Pentium IV 3.0Ghz desktop processor with 512Mb of RAM and no hardware acceleration, such as the use of GPU or SIMD instructions. Some images are depicted in Figure 6.

Table 2 presents results from visualizing several different isosurfaces for various practical datasets, using a screen resolution of 512×512. The isosurfaces were considered to have an opacity value of 100%.

The indicated times are averages of several computed frames, from different viewer angles and positions. The viewing angles were chosen in such a way that there was always neither a perpendicular nor a parallel arrangement between the viewing vector and each of the three main axes. This choice of orientation corresponds to the worst case scenario. Indeed, because we use the ray-isosurface intersection derived at [PSL*98], when the rays are perpendicular or parallel to one of the three principal axes, the resulting polynomial lowers its degree from three to two or even to one.

The times reported here are good at the light of current state of the art of pure software algorithms for isosurfacing. Referring to the times reported by Wald et al. [WHFS05], using no hardware acceleration in terms of GPU or SIMD extensions, it was reported an average of 3.4 and 3.0 frames per second for the bonsai and the aneurism dataset, respectively. In our case, Table 2 indicates 2.5 to 5.0 frames per second for the bonsai dataset and 5.79 to 7.54 frames per second for the aneurism dataset. Nevertheless we acknowledge that such direct comparisons may not be totally fair. Indeed, the hardware used is different (as they used a single dual-1.8 GHz Opteron 246 PC with 6 Gb of RAM) and there is no indication of isovalues [WHFS05].

In Table 3 we have considered several isosurfaces with different opacity values but in the same frame. It highlights the correct implementation of the algorithm regarding the visualization of several isosurfaces with different opacity values.

© The Eurographics Association 2007.

Figure 6: *From left to right, the* aneurism, bonsai, visible male head *and* stent *datasets. The three images with opaque isosurfaces are from tests mentioned in Table 2, whereas the rightmost image with multiple isosurfaces is from Table 3.*

Table 2: *Visualization of opaque isosurfaces, one at a time, using a screen resolution of 512x512. The times are averages of several computed frames, from different viewing orientations.*

DataSet	Resolution	Number of Cells	Isovalue	Number of Active Cells	Time per Frame (seconds)
Aneurism	256x256x256	16,581,375	56	119,698	0.1725
			120	79,282	0.1325
Bonsai	256x256x256	16,581,375	51	310,093	0.4000
			74	212,867	0.2575
			150	279,648	0.2000
Stent8	512x512x174	45,173,933	90	1,236,114	1.3875
			138	1,658,965	1.8925
			153	1,004,858	1.3225
			173	443,093	0.7300
Visible Male Head	126x256x256	8,128,125	50	278,856	0.3475
			79	336,651	0.2825

6. Conclusions and Future Work

In this paper we have presented a new hybrid algorithm for isosurface visualization. Besides the visualization stage itself, where we use raytracing to achieve an image quality comparable to those from the usual direct surface rendering methods, our approach also uses isosurface extraction. This is a well established concept for polygon rendering methods. With this general approach, we are able to avoid empty ray traversal. Actually, empty ray traversal is taken as the bottleneck for direct surface rendering algorithms [PSL*98, WHFS05]. Moreover, it is bound to continue being problematic as the size of real life volumes tends to increase, so potentially increasing the number of cells that do not intersect an isosurface.

The algorithm delivers competitive figures regarding real time performance, when comparing to other pure software methods under similar conditions. We use a caching structure resembling the famous Russian stacking dolls. With this data structure, we correctly process all the active cells for an isosurface considering any viewing orientation. As for the memory overhead, it requires only one copy of the active cells. Worth to mentioning that, in the field of direct volume rendering using 2D textures [RSEB*00], three copies of the volume need to be maintained in order to achieve a correct rendering order from any viewing orientation. The *Matryoshka* structure can be considered for volume rendering methods in general.

We are planning to provide occlusion tests at the level of the layer: if one layer is completely occluded in the image plane, then all the interior layers can be discarded. This will improve the performance as it may avoid the processing of some cells. On the other hand, any advances in hierachical occlusion maps [MJC02], particularly if adjusted to our concept of layers, can be very useful.

While pursuing our goals for the visualization algorithm, we also have investigated new methods for isosurface extraction. We have proposed two new versions of interval trees, which significantly decrease the memory overhead associated with such type of data structure. The choice of which version to use is really application dependent. Interval tree with grouping yields to less memory overhead, whereas interval tree with both grouping and ordering delivers the lowest query times. As the volume size increases, these lowest query times are quite comparable to the query times from a normal interval tree.

© The Eurographics Association 2007.

Table 3: *Visualization of isosurfaces with different opacity values but in the same frame. Further tests conditions are the same as in Table 2.*

DataSet	Isovalues / Opacities				Time per Frame (seconds)
Stent8	90 / 20%	138 / 40%	153 / 40%	173 / 100%	10.783
Visible Male Head	50 / 20%	79 / 100%	-	-	0.870

In the future we will use very large datasets to extend the evaluation of the algorithm. Furthermore, we will investigate if some of these ideas can be implemented at GPU level.

References

[BvKOS00] BERG, VAN KREVELD, OVERMARS, SCHWARZKOPF: *Computational Geometry Algorithms and Applications*, second ed. Springer, 2000.

[CMPS96] CIGNONI P., MONTANI C., PUPPO E., SCOPIGNO R.: Optimal isosurface extraction from irregular volume data. *Symposium on Volume Visualization'96* (1996), 31–38.

[Ede80] EDELSBRUNNER H.: Dynamic data structures for orthogonal intersection queries. Technical Report F59, Inst. Informationsverarb, Tech. Univ. Graz, Graz, Austria, 1980.

[Erl] http://www9.cs.fau.de/persons/roettger/library/.

[LB03] LOPES A., BRODLIE K.: Improving the robustness and accuracy of the marching cubes algorithm for isosurfacing. *IEEE Transactions on Visualization and Computer Graphics 9*, 1 (2003), 16–29.

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics (Proc. of SIGGRAPH) 21*, 4 (1987), 163–169.

[LSJ96] LIVNAT Y., SHEN H.-W., JOHNSON C. R.: A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics 2*, 1 (1996), 73–84.

[MJC02] MORA B., JESSEL J.-P., CAUBET R.: A new object-order ray-casting algorithm. In *VIS '02: Proceedings of the conference on Visualization '02* (Washington, DC, USA, 2002), IEEE Computer Society.

[MKW*04] MARMITT G., KLEER A., WALD I., FRIEDRICH H., SLUSALLEK P.: Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In *Proceedings of Vision, Modeling, and Visualization (VMV)* (2004), pp. 429–435.

[Nat94] NATARAJAN B. K.: On generating topologically consistent isosurfaces from uniform samples. *Vis. Comput. 11*, 1 (1994), 52–62.

[NH91] NIELSON G. M., HAMANN B.: The asymptotic decider: resolving the ambiguity in marching cubes. In *VIS '91: Proceedings of the 2nd conference on Visualization '91* (Los Alamitos, CA, USA, 1991), IEEE Computer Society Press, pp. 83–91.

[NMHW02] NEUBAUER A., MROZ L., HAUSER H., WEGENKITTL R.: Cell-based first-hit ray casting. *Proc. Symp. Data VIsualization 2002* (2002), 77–ff.

[PPL*99] PARKER S., PARKER M., LIVNAT Y., SLOAN P.-P., HANSEN C.: Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics 5*, 3 (1999).

[PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.-P.: Interactive ray tracing for isosurface rendering. *proc. IEEE Visualization '98* (1998), 233–238.

[RSEB*00] REZK-SALAMA C., ENGEL K., BAUER M., GREINER G., ERTL T.: Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '00* (2000), Addison-Wesley Publishing Company, Inc., pp. 109–118,147.

[Sch90] SCHWARZE J.: Cubic and quartic roots. 404–407.

[SHLJ96] SHEN H., HANSEN C., LIVNAT Y., JOHNSON C.: Isosurfacing in span space with utmost efficiency. *Proc. of IEEE Visualization'96* (1996), 287–294.

[UO93] UDUPA J. K., ODHNER D.: Shell rendering. *IEEE Comput. Graph. Appl. 13*, 6 (1993), 58–67.

[vol] http://www.volvis.org/.

[WHFS05] WALD I., HEIKO FRIEDRICH GERD MARMITT P. S., SEIDEL H.-P.: Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics 11*, 5 (2005).

© The Eurographics Association 2007.