State-of-the-art: Frame-to-frame coherence in volume rendering

S. Grau

Volume data describe the internal features of a solid object. A volume can be represented as a regular 3D array of data values, also called voxels that store one or more property values such as density, activity, or, in fluid dynamics datasets, velocity. The voxels are characterized by their position in the 3D grid. In some datasets, called time-varying datasets, the property values evolve through time.

This document is a review of the state-of-the-art in time-varying volume data rendering. First, in Section 1, we review two of the most popular volume rendering methods for static data, specifically ray-casting and splatting. We also survey the strategies that have been proposed to speed up these methods.

In Section 2, we analyze the existing time-varying rendering methods for polygonal models as well as for volume data. Finally, we present the conclusions in Section 3.

1 Static Data Volume Rendering

1.1 Basic techniques

There are different strategies to render static data, that can be grouped in two different families [59]:

- Indirect Volume Rendering (IVR). These algorithms convert the volumetric data into a set of polygonal isosurfaces and, next, render them with polygon rendering

hardware. The most common algorithm to extract an isosurface from a volume is Marching Cubes [49] [62].

- Direct Volume Rendering (DVR). The volume datasets are directly rendered without intermediate conversion step. The most used DVR techniques are: Ray-casting, Splatting, Shear-warp and 3D hardware-based texture-mapping [61].

IVR strategies are out of the scope of this work. We next briefly describe the four DVR basic approaches.

1.1.1 Ray-casting

Ray-casting has seen the largest body of publications over the years. It launches one or more viewing rays for every pixel and samples the rays through the volume. At each sample, the property value and gradient at each sample point are interpolated, then classification and shading are performed. The samples intensities and opacities are composited in order to obtain the corresponding pixel intensity value (see Figure 1). This basic method has been implemented with several variations. For instance, the Maximum Intensity Projection (MIP) [25] [33] [17], a very popular technique to outline blood vessels, does not actually compose and blend colors along the ray, but as its name indicates, it simply computes and projects the maximum property value. Ray-casting can be implemented completely via software, but, as explained in Section 1.3.2, recently hardware-based implementations of this technique have been designed [42] [78].

1.1.2 Splatting

The splatting algorithm was proposed by Lee Westover [92]. It considers the volume as an array of overlapping kernels that are projected into the screen plane in order to compose the image (see Figure 2). Splatting gains its speed by exploiting the similarity

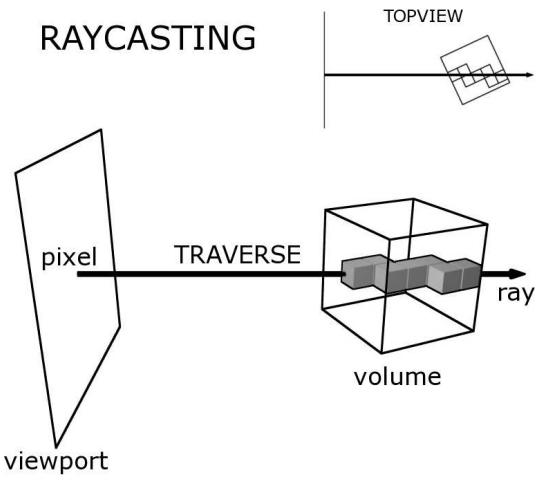

Figure 1: Direct Volume Rendering Techniques: Ray-casting strategy.

of the kernel's projection. In orthographic views, all the kernels have the same projection or *footprint*. Thus, the footprint can be computed once, in a pre-process, stored as a look-up-table and used for the projection of all the voxels. However, in perspective views, the footprints must be distorted according to the distance of the voxels to the observer.

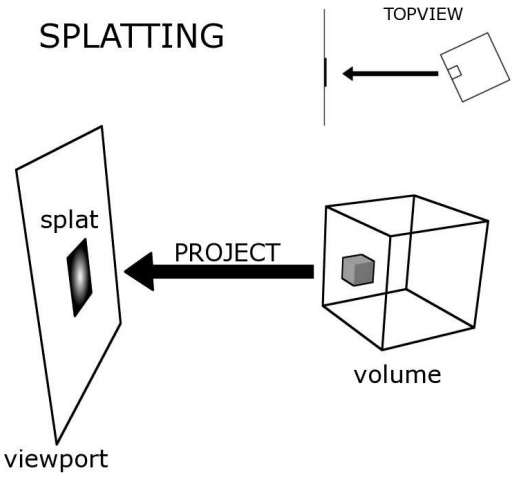

Figure 2: Direct Volume Rendering Techniques (II): Splatting strategy.

In the original approach of the algorithm, all the voxels are splatted directly in the image. This is why the algorithm is known as *composite-every-sample* (CES). However, this method may cause color bleeding and sparkling artifacts because the visibility or-

dering of the splats is imperfect. To correct this error, Westover [93] proposed the *object-space sheet-buffer splatting* (OSS) that splats the voxels slice-by-slice into sheet planes of the voxel model most parallel to the image plane and composites each sheet to the final image. This approach corrects color bleeding but it introduces noticeable popping up artifacts when the camera moves around the volume, because the sheet planes chosen change abruptly. Mueller and Crawfis [65] provided a solution to this problem that also enhances the approximation of the light transport inside voxels: the *image-space sheet-buffer splatting* (ISS). In this approach, the sheet buffers are parallel to the image plane. Therefore, voxels can contribute to more than one sheet. Different footprints corresponding to different intersections of the voxels with the sheet slab must be computed. When a voxel is splatted into a sheet plane, the proper footprint is chosen according to a fast indexing scheme. In the *image-space sheet-buffer splatting* [66], sheet buffers can be composed Front-to-Back (FTB) in order to apply early splat elimination by subdividing the image into small tiles and avoiding to splat voxels that cover tiles that have already reached the maximum opacity. The detection of opaque tiles is efficiently performed using a hardware assisted opacity convolution filter.

Figure 3 illustrates the general pipeline of three splatting strategies composed of six main operations:

- Access to the voxel Value (V)
- Object-space Gradient computation (G)
- Shading (S)
- Viewing geometrical Transformation of a voxel (T)
- Voxel Splatting with or without α-blending (SP)
- Bucket Insertion (BI)
- Buffer Composition (BC)

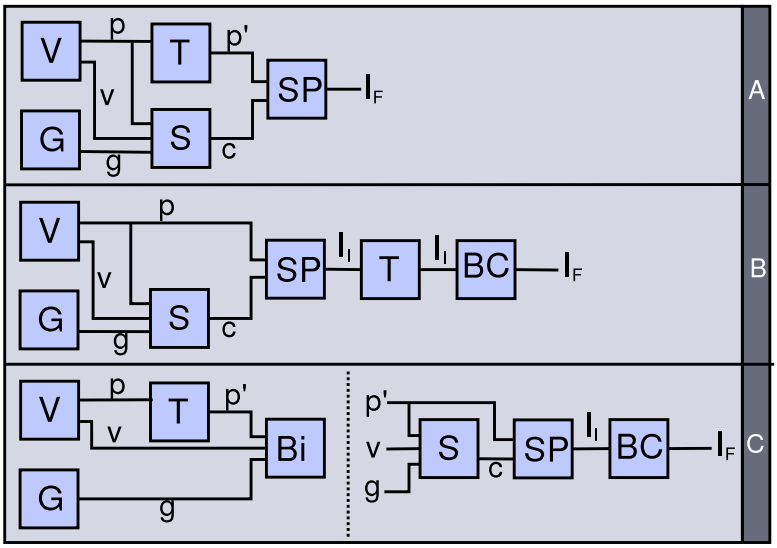

Figure 3: Rendering pipelines of the three splatting strategies from top to down: Composite-Every-Sample (CES), Object-space Sheet-buffer Splatting (OSS) and Image-space sheet-buffer Splatting (ISS).

Observe that the gradients are computed in object space. However, as suggested by Neophitou et al.[68], pixel gradients can be computed in image-space.

As ray-casting, splatting can be totally software-based, but, recently, hardware-driven implementations of parts of the pipeline [68] have been proposed.

1.1.3 Shear-warp

The Shear-warp approach was proposed by Lacroute and Levoy [43]. It employs a clever volume and image encoding scheme, coupled with a simultaneous traversal of volume and image that skips opaque image regions and transparent voxels. It achieves this by performing a run-length encoding (RLE) compression of the volume to allow a fast streaming through the volume data, and rendering using a simultaneous object-order and image-order traversal to rapidly splat entire slices of the volume onto the image plane. The Figure 4 shows the pipeline of this method. First, the volume is sheared and resampled in object space. Second, the resampled slices are composited in front-to-back order. Finally, the intermediate image is warped and resampled to the

final image.

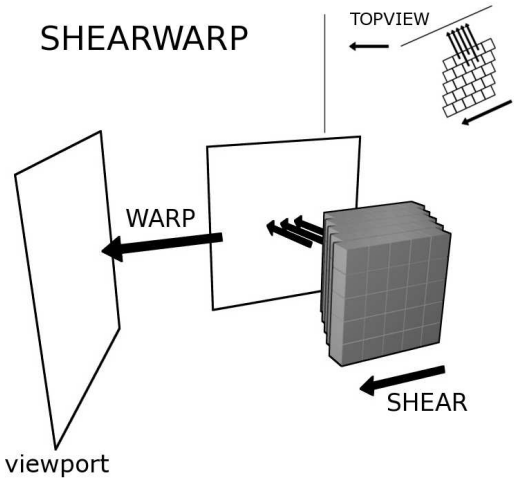

Figure 4: Direct Volume Rendering Techniques (III): Shear-warp method.

The advantage of this technique is that the the volume data scanlines and the intermediate image scanlines are always aligned and, thus, the necessary projections are straight forward and quite efficient. Furthermore, the algorithm gains speed and efficiency since the filter operations are performed in 2D. Shear-warp rendering is the fastest software-based DVR generating images at interactive frame rates. However, the produced artifacts of the resampling filters used, the required high among of memory needed and the advent 3D texturing and programmable shaders have considerably reduced its use.

1.1.4 3D Texture-mapping

The launching of the SGI Reality Engine [1] in the early nineteens made possible the use of three-dimensional texture-mapping to render volume. Neumann and Cullip [18] and Cabral et al. [12] proposed to load the volume into texture memory, to compute and compose a set of polygonal slices parallel to the viewplane (see Figure 5). The speed of the method comes from the hardware-driven rasterization of the slices.

Until the advent of programmable shaders, the main concern with this technique was to provide it with shading capabilities within the graphics pipeline. Van Gelder et al. [26] proposed to store in the 3D texture indexes to a look-up table rather that color

values. Their approach, however, was not hardware-supported and required the textures to be reconstructed on classification and lighting changes. Westermann et al. [90] and Dachille et al. [19] proposed to store gradient values as well to density values in order to render shaded isosurfaces. Meissner et al. [58] extended this approach by integrating in the rendering pipeline diffuse semi-transparent shading. Rezk-Salama et al. [74] proposed to use register combiners instead of matrix multiplication. Engel et al. [22] used multi-texture interpolation in order to reduce the number of required slices. Finally, Meissner et al. [57] integrated lighting models in 3D texture-mapping for PC graphics hardware.

Modern graphics cards have the ability to replace parts of the rendering pipeline with small programs called shaders. There are three kinds of shaders: vertex, geometry and fragment shaders. Vertex shaders control things like vertex position, normals, texture coordinates. Geometry shaders are used to generate extra vertices inside polygons. They are only available on the most modern graphics cards. Fragment shaders are basically used to draw the interior of a polygon. They operate on individual fragments and have full control over pixel color. They handle things such as texturing, per-pixel lighting, and advanced effects.

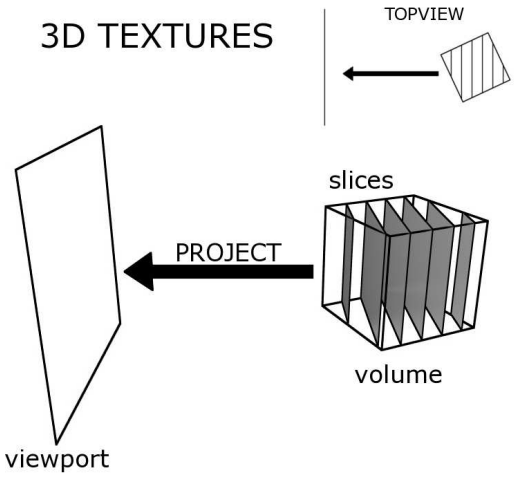

Figure 5: Direct Volume Rendering Techniques (IV): 3D texture-mapping method.

1.2 Accelerating Volume Visualization

The major problem identified in volume rendering is the time that it requires. Many efforts have been put in reducing these time requirements, that can be classified into 3 main approaches:

- using hardware specifically designed for volume rendering applications
- exploiting modern graphics hardware facilities to speed up rendering
- designing specific strategies such as space-leaping, to speed up the computations of classical image-order and object-order methods

1.2.1 Specific hardware

Special purpose hardware for volume rendering has been proposed by various researchers, but only a few machines have been implemented. Some examples are: Cube-4 volume rendering architecture developed at SUNY Stony Brook [71]; VIRIM, built by the University of Mannheiming (Germany) that implements parallel ray-casting [29]; VIZARD and VIZARD II, built by the University of Tbingen (Germany), implements true perspective ray-casting [60]; VolumePro implements parallel hybrid ray-casting algorithm [61]. This kind of approaches are out of the scope of this work.

1.2.2 Exploiting graphic cards

The extended use of videogames in low end platforms benefits the fast development of the graphic cards features. Nowadays, the use of these characteristics for other uses is one of the most important lines of research in computer graphics. In volume rendering the focus on developing new algorithms is beginning to shift towards higher quality rendering and additional functionality instead of simply higher performance implementations of the traditional graphics pipeline.

Graphics libraries like OpenGL (www.opengl.org), and its extensions provide access to advanced graphics operations in the geometry and the rasterization stage and therefore, allow for the design and implementation of completely new classes of rendering algorithms.

There are various approaches that make extensive use of graphics hardware for rendering volumetric data sets. Three relevant approximations are mentioned : Hardware support for the interactive rendering of complex 3D polygonal scenes consisting of directly lit and shaded triangles have become widely available (it is used in IVR [7]), hardware-based implementations of the classical volume rendering methods, namely ray-casting and splatting and the use of 3D textures in DVR applications exploiting the processing power and functionality of the rasterization and texture subsystem of advanced graphics hardware [26] [90] [58].

1.3 Accelerating Ray-casting

1.3.1 Software-based acceleration of Ray-casting

Most of existing methods for speeding up Ray-casting rely on one or more of the following principles [61]:

- Pixel-space coherency: There is a high coherency between pixels in image space. That is, it is highly probable that between two pixels having identical or similar color there is another pixel having the same (or similar) color.

- Object-space coherency: The extension of the pixel-space coherency to 3D states that there is coherency between voxels in object space.

- Inter-ray coherency: There is a great deal of coherency between rays in parallel viewing, that is, all rays, although having different origin, have the same slope. Therefore, the set of steps these rays take when traversing the volume are similar. This property is used in as for instance in the template-based method [100][98].

The acceleration of Ray-casting on the basis of these properties consists basically of these strategies: adaptive ray sampling, early termination, space-leaping and fast ray-sampling computation. We next survey these techniques.

Adaptive ray-sampling has been used in ray-casting to exploit pixel coherency as well object coherency. In image space adaptive supersampling speeds up antialiasing [46]. Along the rays, object coherency can be used to sample the volume at different frequency levels. In a method introduced by van Walsum et al [83], the ray starts sampling the volume at low frequency, taking large steps between sample points. If a large value difference is encountered between two adjacent samples, additional samples are taken between them.

Early termination techniques can be applied if the composition is Front-to-Back (FTB). They are aimed at avoiding sampling regions of the volume model that cannot contribute to the image because they project onto regions of the image space where the opacity has already reached a value close to unity. Early termination can be very easily implemented in a ray-casting approach, since it simply consists of stopping the integration along the rays when the corresponding accumulated pixel opacity is 1.0. This strategy called *early ray termination* has been used by Levoy [45] and Danskin and Hanrahan [20] among others.

Finally, **space-leaping** consists of skipping empty regions of the model in order to reduce the computations. Space-leaping can also be applied on classified sets to skip regions non selected for rendering. The passage of a ray through the volume is two phased. In the first phase the ray advances through the empty space searching for an object. In the second phase the ray integrates colors and opacities as it penetrates the object (in the case of multiple or concave objects these two phases can repeat). Commonly, the second phase involves one or a few steps, depending on the object's opacity. Since the passage of empty space does not contribute to the final image it is observed that skipping the empty space could provide significant speed up without

affecting image quality.

The main space-leaping techniques for ray casting are :

- Use of hierarchical structures. The hierarchical representation (e.g., octree) decomposes the volume into uniform regions that can be represented by nodes in a hierarchical data structure. An adjusted ray traversal algorithm skips the (uniform) empty space by maneuvering through the hierarchical data structure [45]. Hierarchical data structures used to skip empty space during rendering, such as kd-trees [79] and octrees [45] [44] [95], present two major drawbacks. First, the tree traversal causes a cost overhead. Next, the error associated to the nodes is a global parameter, therefore, the data structure does not provide a local control of the error in a specific region.

- Use of bounding boxes. When a volume consists of one object surrounded by empty space, a common and simple method to skip most of this empty space uses the technique of bounding-boxes. The object is surrounded by a tightly fit box (or other easy-to-intersect objects such as spheres). Rays are intersected with the bounding object and start their actual volume traversal from this intersection point as opposed to starting from the volume boundary [4].

- Use of distance maps. The proximity-clouds method [14] is based on the extension of this idea even further. This method computes, in a preprocessing stage, for each empty voxel, the distance to the closest occupied voxel [24]. This strategy has been exploited by Zuiderveld et al. [103] for multimodal rendering.

Finally, **Fast ray sampling computation** can be used such as 3D discrete digital analyzer, in order to simplify the point sampling computations, to reduce the intra-voxel sampling and to avoid aliasing. In particular, Yagel et al. [98] have proposed to utilize one fast and crude line algorithm in the empty space (e.g., 3D integer-based 26-connected line algorithm) and another, slower but more accurate (e.g., 6-connected

integer or 3D DDA floating point line algorithm), in the vicinity and interior of objects. Thus, they do not actually skip voxels but they skip samples inside empty voxels.

1.3.2 Hardware-based acceleration of ray-casting

Recently, GPU-based implementations of ray-casting for structured and unstructured volume grids have been proposed [42],[78] [89]. Specifically, for regular grids, Kreeger and Westerman [42], proposed a multi-pass approach. They first render the front faces of the volume bounding box as a 2D texture storing the coordinates of the vertices. Then, they do the same task for the back faces, but they actually store the ray direction in the 2D texture. They compute the ray direction by normalizing the difference vector composed of corresponding front and back vertices. They store the ray length, i.e. the distance between those vertices in the α-channel of the texture. Then, the technique proceeds in various passes, performing M steps along the ray at each pass and storing the results in a texture which is re-used in the next step. This technique performs early ray termination, by using an auxiliary Z-Buffer and writing in that buffer maximum Z values at pixels having already reached the maximum opacity. In addition, it performs space-leaping by using an auxiliary 3D raster texture of $1/8$ the size of the original volume that encodes the minimum and maximum value of the corresponding block in the R and G channels.

More recently, Stegmaier et al. [78] exploit more recent graphics capabilities such as branching and looping in fragment level shading programs. Their strategy performs ray-casting in a single pass.

1.4 Accelerating splatting

1.4.1 Software-based acceleration of splatting

The early termination strategy for splatting is called *early splat elimination*. It is is more complicated than early ray termination since in order to avoid to splat a voxel

all the pixels of its splat should have reached opacity one. Checking all the pixels is very time consuming. Therefore, taking profit of *pixel coherency*, image space quadtree subdivision schemes have been applied [56] and, in the context of view-aligned splat, tile-based checking with convolution filtering [66]

One of the major advantage of splatting is that only relevant voxels must be splatted and empty and non-selected voxels can be skipped, so space-leaping is naturally embedded in splatting. This idea was first suggested by Yagel et al. [99] for rendering Computational Fluid Dynamics (CFD). They suggested to construct a *fuzzy set* composed by an array of planes of the model and, for each plane, a list of voxels with their associated coordinated in the plane and their value. Crawfis [15] introduced the idea of the *ListSplat*, a list of isosurface voxels that can be splatted directly without depth sorting because they are supposed to all be a homogeneous color. Mueller et al. [66] enhanced the efficiency of the view-aligned sheet-buffer splatting by organizing the selected voxels in buckets, each one corresponding to a sheet-buffer. The selection of the voxels for their insertion in the buckets is fast, based on a binary search in a per-value ordered list of voxels similarly to the work of Ihm et al. [37]. The RTVR system [64] uses an intermediate array of slice-sorted *RenderLists* for each object that stores the voxels of each slice which are relevant for rendering, and those that, not being empty, may be clipped. In addition, it requires to store the position of the non-empty samples, via a de-referencing mechanism and thus, it sacrifices the benefit, in terms of memory requirements, of the implicit spatial arrangement of the voxel model. The *RenderLists* are used in the same way in the *Two-level rendering* proposed by Hauser et al. [30]. More recently, Orchard and Möller [70], proposed to use a list of adjacency data structure, such that each non-empty voxel in a scan list is linked to the next non empty voxel in the scan-line.

Kilthau and Möller [39] proposed to use run-length encoding (RLE) of the volume in order to skip empty voxels. Run-length encoding consists of representing the volume

array as a list of codes composed by the voxel value and the number of voxels that share this value. The shear-warp algorithm is precisely based on a double run-length encoding (RLE) of the voxel array and the image scan-line [43]. Kilthau et al. [39] propose to to construct 24 RLE replications of the volume, which allows them to orderly traverse the the volume according to any of the 48 orders. Run-length encoding consists of representing has been developed also in the context of object-order splatting for multimodal data [23]. Its major drawback is that it requires to traverse the voxel model in the encoding order therefore, it is not suitable for ray-casting and for view-aligned splatting and texture-mapping. The main drawback to these two last approaches is their storage overhead.

Finally, in Li et al. [47], propose to use disjoint bounding boxes of specific features. Although the technique is primarily intended at treating 3D texture mapping, it is also suitable for splatting. Since the boxes are disjoints, their relative order must be preserved using an orthogonal BSP tree. The boxes are completely full of the region therefore, they provide an efficient access to the region. On the opposite, the number of boxes is very high. For hybrid shading, the authors propose to separate into specific boxes zero-gradient and non-zero gradient voxels, which avoids applying the gradient computation to the former group. Another advantage of this strategy is that it benefits from hardware acceleration through 3D texture-mapping. However, the main drawback of the boxes approach is that the classification is based on the spatial organization rather than on the property space. However, although semantic regions often correspond to connected sets of voxels that may be clustered in spatial regions, sometimes selected data are spread in the volume, yielding to an increase of the boxes number.

1.4.2 Hardware acceleration of Splatting

Many efforts have been done in accelerating splatting using hardware. One of the first proposed method [44] [94] consists of approximating the splat by a collection of poly-

gons, thus taking profit of the hardware-supported polygon rendering pipeline. Crawfis and Max [16] replaced the polygons by a 2D texture map. These approaches were tested in *composite-every-sample* traversals and orthographic projections in which only one footprint is necessary. Huang et al. [35] argued that *image-space sheet-buffer splatting* requires at least 128 footprint sections, which supposes over than 8MB texture maps storage. For radially symmetric splats, they propose to use a less-memory consuming one-dimensional table that holds the values of the splat along a radial line from the splat center. Moreover, they explore directly copying into the image the block of pixels of a 2D footprint using BitBLT, but conclude that the image quality of this strategy is low. More recently, Xue and Crawfis [97] proposed two splatting strategies that work on the GPU. The first strategy consists of using a vertex shader program to generate and render quadrilaterals centered around the voxels center. This strategy works on previous generation hardware. In addition, it requires sorting the voxels along the viewing direction and it has high memory requirements. The second strategy, point-convolution rendering, first projects all the voxels as point primitives into an off-screen P-Buffer with additive blending. Next, the GL convolution flag is activated and a texture is copied from the P-Buffer using *glTexSubImage2D* such that each texel is a convolution between the P-Buffer pixel and the kernel filter. This strategy is very efficient in terms of computational cost but it only renders x-ray style images for orthographic views. Very recently, Vega-Figueroa et al. [84] propose to use *Point Sprites* to render neurovascular data. This reduces to one point per voxel the geometric processing tasks instead of the four-points needed for the quadrilaterals. This idea is also exploited in the GPU-based implementation of the *image-space sheet-buffer splatting* proposed by Neophitou and Mueller [68]. In addition, this paper proposes to use an OpenGL PBuffer object to store the buffers. It first splats onto an auxiliary buffer the density value of all the voxels of a slice using textured point sprites. Then, it classifies and shades all the pixels of the buffer using a fragment shader that computes the gradient vectors at the pixels

on the basis of their density central difference. Finally, it composes the buffer into the final image. The authors use the early z-rejection test to eliminate empty-space pixels and those that are already opaque before the fragment processing. More recently, the same authors [69] propose to store the bucket's voxels in a texture in order to speed-up data transfer between CPU and GPU. A comparison between different hardware and software-based optimizations of splatting has been done by Vergés et al. [85].

2 Time-varying rendering

We first classify the different parameters that change at each instant in a time-varying scene and we define the property of *frame-to-frame coherence* which is used in time-varying algorithms in order to speed up computations. We next survey briefly the use of this property in polygonal scenes and describe existing time-varying volume rendering algorithms.

2.1 Basic concepts

Different elements of a scene can vary through time: the objects, their internal properties, the camera and rendering parameters such as transfer functions and lighting conditions. The strategies that can be used depend on theses cases. For clarity, we will distinguish each case using the following nomenclature.

- **Volume animation:** when the volume datasets move. This movement can be an affine transformation of translation and rotation of all the model or a non-uniform displacement of the sample points. The former case happens whenever the volume is part of a huger scene, composed of other objects, either polygonal or volumetric [41]. In the later case, the volume is actually deformed.

- **Time-varying animation:** when the volume is static but the properties inside the cells vary. Two typical applications of this type of scenes are (i) a temporal

series of SPECT of a patient's brain and (ii) the simulation of a fluid flowing in a fixed section of a channel.

- **Fly-through navigation:** when the volume dataset is static and its properties constant but the viewer's position and direction varies though time, because it navigates through the data. Typical examples of fly-through navigations are the virtual cateterism [72], virtual colonoscopy [34] [88] and bronchoscopy [8], [63]. Some of these papers perform the navigation through surface models previously extracted from volume datasets, while others [11] actually navigate through the volume. The major problem addressed in this type of navigation is how to efficiently perform visibility culling and to bring into memory the portions of volumes that fall in the current viewing frustrum. In this report, we do not address fly-through navigation.

- **Fly-around navigation:** when the volume dataset is static and its properties constant but the viewer's position and direction varies, because it moves around the volume, without entering inside it [101].

Obviously, these cases can be combined. As an example, if the viewer moves around a volume whose properties vary through time, we will talk about a *Fly-around navigation of time-varying volume data*. If the viewer navigates inside the same volume, it will be a *Fly-through navigation of time-varying volume data*. If the volume moves and its properties vary, we will talk about a *Volume animation of time-varying data*.

We do not enclose in this taxonomy the case in which only user-defined rendering parameters such as lighting conditions and transfer functions change. This has much to do with interactivity than actually animation and its variants.

In most of these cases, rendering consists of generating a sequence of images of the volume at different instants throughout a period of time. Inspired on the early photographic methods of Marey and Muybridge, this technique performs an integration through time

and produces a single view that captures the essence of various key-instants of the sequence. It is suitable for time-varying voxel models. The Chronovolume is a voxel model such that every voxel is computed by integrating all the voxel values throughout time for a given transfer function. The chronovolume is rendered as a regular volume in order to produce 2D images.

2.2 Frame-to-frame coherence

If we inspect an image in a sequence, we will find that large areas in do not change at all in relation to the previous image, or have changed very little. The significant differences between frames is generally concentrated in a few restricted regions, whose total area is small compared to the entire image.

As mentioned by Sudarsky [80], the similarity between consecutive images in an animation sequence is called *temporal coherence*. In fact, the similarity is not just between the images themselves, but also between the scene model states at consecutive frame times: the object forms and locations, surface properties, light sources and viewpoint all change gradually, and are almost identical at close points in time. This kind of similarity is also called *temporal coherence*. To distinguish between the two types of coherence, the similarity between the images themselves is called *image-space temporal coherence*, while the latter is commonly known as *object-space temporal coherence*. Both kinds of coherence may be utilized to save computations when rendering images as part of an animation sequence.

2.3 Frame-to-frame coherence in surface-based visualization

Starting from the early work of Matsushita [55] and Hubschman and Zucker [36], many attempts have been done to speed up rendering of time-varying surface-based scenes. Those algorithms works only for static scenes and moving viewer with convex, non-intersecting, finite polyhedra. Reprojection techniques can be applied that avoid the

cost of recomputing intersections [5]. A recent paper that addresses reprojection is done by Havran [31]. Tost [82] described a hidden surface removal algorithm, applicable to general polyhedral scenes.

When the scene changes but the viewpoint is static, ray-casting can be used to generate simultaneously multiple frames. Chapman et al. [13] consider a single pixel and process its value for all frames in an animation. *Pixel coherency* between successive frame is exploited by not tracing the ray at each frame but at the extreme of time intervals by Badt [5]. If the difference between the pixel values is lower than a threshold, the pixel value is simply replicated during all the interval. The pixel can also be computed by interpolation [52]. Recently, some of these previous ideas have been re-used in an efficient spatio-temporal architecture that computes multiple frames at once extending the reprojection techniques to support camera motion [32].

Another strategy consists of codifying the scene into 4D structures. Glassner's [27] hexatree is an extension of the octree to the 4th dimension. Rays are recursively intersected with the nodes of the hexatree, which avoids reducing the number of intersections per ray and per frame. Grller and Purgathofer [28] extend Arvo and Kirk's ray classification method [3] to utilize temporal coherence in animation of CSG models.

An alternative idea to separate the static part of the scene from the dynamic one, trying to restrict the frame computations to an update. This poses several difficulties when global illumination is applied since the illumination of the static part depends also on the moving objects. This problem has been addressed by Wald et al. [86], Besuievsky et al. [9] and Martin et al. [53], among others.

Frame-to-frame coherence has also applied for temporal aliasing motion blur computation [38].

2.4 Time-varying Direct Volume Visualization

Previous work on direct time-varying volume rendering typically fall into categories: one that treats separately the time dimension from the spatial dimensions [101] [77] [76] [2] [91] [50], and the other (4D rendering) that treats time-varying data as a special case of an n-D model [6], [21][67] [96]. Our approach belongs to the first group.

Papers of the first group exploit temporal coherence in different ways. Most of them focus on the ray-casting strategy.

2.4.1 Temporal coherence in volume ray-casting

Yagel and Shi [101] propose a frame-to-frame coherent ray-casting that stores in a *C-Buffer* the coordinates of the first non-transparent voxel encountered by the ray emitted at each pixel. If the light conditions or the transfer function changes in successive frames, the ray sampling can start at this location and skip the previous empty voxels. Moreover, the *C-Buffer* can be re-used for reprojection if the camera rotates. Actually, Wan et al. [87] found that the original point-based reprojection method can create artificial hole pixels, that can be corrected using a cell-reprojection scheme. Both approaches speed up ray casting computations when the camera or the transfer function change. However, when the property varies inside the voxels and the empty voxels change along time, the *C-Buffer* must be recomputed. The reprojection technique has also been used to track points across frames in order to reduce temporal aliasing [54].

Recently, Klein et al. [40] use these ideas in a GPU-based implementation of ray-casting [42] [78]. They implement the *C-Buffer* as a render target to store the *hit position*, an, whenever the camera moves, they reproject the hitpoints stored in this render target using OpenGL viewing matrices. The holes in the reprojected image are avoided by using enlarged points. The authors also propose a selective super-sampling object space antialiasing technique.

Shen and Johnson [77] focuses on exploiting ray coherence when the property values

inside the voxels change along time and the camera remains static. Given the initial data sets, this method constructs a voxel model for the first frame and a set of incremental models for the successive frames, composed of the coordinates of the modified voxels and their values. The first frame is computed from scratch. The next frames are computed by determining which pixels are affected by the modified voxels of the corresponding incremental file, updating the voxel model and recasting only the modified rays. This strategy produces a significant speed up of the animation if the incremental files are small, i.e., the number of modified voxels is low. However, the incremental files do not keep the spatial ordering of the voxel models. Therefore, the method is not suitable to visualize sub-models or specific features in a model. In a recent paper, Liao et al. [48] propose an improvement of this technique consisting of computing an additional differential file, called SOD (Second Order Differential file) that stores the changed pixels positions. At each frame, the rays are either computed following Shen et al.'s strategy [77] or using the SOD, i.e. accessing directly to the modified pixels, avoiding the cost of projection of the modified voxels.

Reinhard et al. [73] address the I/O bottleneck of time-varying fields in the context of ray-casting isosurfaces. They propose to partition each time step into a number of small files containing a small range of iso-values. They use a multiprocessor architecture such that, during rendering, while one processor reads the next step time, the other ones render the data currently in memory. Their results show that partitioning data is an effective out-of-core solution. Binotto et al. [10] propose to compress highly coherent time-varying datasets into 3D textures using a simple indexing scheme mechanism that can be implemented using fragment shaders. Youmesy et al. [102] accelerate data load at each frame using a differential histogram table that takes into account data coherence. Ma et al. [51] explore the use of a BON (Branch-On-Need) octree [95] for time-varying data. The construction of the tree consists of three steps: quantization of the volume, construction of a BON for every instant of time and merging of the subtrees that

are identical in successive BONs. This data structure is rendered with ray-casting by processing the first BON completely, and only the modified subtrees of the following BONs. To do so, an auxiliary octree, called the *compositing tree*, is constructed, similar to the BON, that stores at each node the partial image corresponding to the subtree. At successive frames, when a subtree changes, its sub-image is recomputed and composited at its parent level in the hierarchy. In addition, a wavelet-based variant of the TSP (WTSP) has recently been published [75]. The author de-correlates the time-varying data into a range of spatial and temporal levels of detail for the purpose of enabling rapid run-time data retrieval, reconstruction, and rendering.

The TSP *Temporal Space Tree* [76] is a spatial octree that stores at each node a binary tree that represents the evolution of the subtree through time. The TSP tree can store partial sub-images to accelerate ray-casting rendering, and it has also been used to speed up texture-based rendering [21]. This structure is particularly suitable for datasets in which most of the volume remains almost static and only specific regions vary through time. Otherwise, the tree can be highly subdivided and only few partial images can be re-used. The extension of these octree-based methods to multimodality would require the datasets to be aligned. In addition, the sub-images would hardly be re-usable, since the fusion of different modalities must be done at the sample level and not at sub-images in order to preserve correct depth integration.

2.4.2 Temporal coherence in splatting

The splatting algorithm proposed by Neophitou and Mueller [67] belongs to the 4-D rendering approach. They use a 4D Boby Centered Cubic (BCC) grid [81] instead of the traditional 4D cartesian Grid (CC) because it provides compression to about 50% of the original size of the models. At an instant, a hyperslice of the 4D model is first computed by interpolation and next, rendered with a view-aligned sheet-buffer splatting. The hyperslice is encoded into an RLE list which is traversed each time the

transfer function or the viewing parameters change in order to toss the voxels into the array of buckets.

2.4.3 Temporal coherence in shear-warp

The shear-warp technique proposed by Anagnostou et al. [2] uses an incremental Run-Length Encoding (RLE) of the volume. Whenever a change is detected over time, the RLE is updated by properly inserting the modified runs in the volume scan-line. In addition, the volume is processed by slabs, recomputing only the modified slabs and compositing them with the unchanged slabs.

2.4.4 Temporal coherence in 3D texture mapping

Finally, Lum et al.'s approach [50] is based on hardware assisted texture mapping. The time-varying volume over a given span of time is compressed using the Discrete Cosine Transform (DCT). Every sample within the span is encoded as a single index. The volume is represented as a set of 2D paletted textures. The textures are decoded using a time-varying palette. In order to keep a constant frame rate, the texture slices re-encoding at the end of each time-span is interleaved. A parallel implementation is also described.

3 Conclusions

The analysis of frame-to-frame coherent algorithms for polygonal and volume scenes leads us to the following conclusions:

- Most of the frame-to-frame coherent algorithms for volume data are based on a ray-casting approach, probably because it is the most easily adaptable to time-varying data.

- Although the relationship between the two types of scenes (polygons and voxels) and their temporal coherence is obvious, as far as we know, it has not been pointed out explicitly in the existing bibliography. We believe that interrelating the two topics can help to extend existing techniques to both types of scenes.

- The use of 4D hierarchical structures has been found in both polygonal scenes [27] [28] and volume scenes [51] [76][21].

- A technique that has been addressed for polygonal scenes [5] [31] as well as for volumetric ones [101] [87] [54] is reprojection. This technique has been used for static volumes and moving camera, in order to skip only empty space. This idea could be extended to fastly re-cast rays into selected regions.

- When the camera is static, pixel coherency has been used in polygonal scenes to separate static objects from dynamic ones and recast rays only in the latter ones [86] [9] [53]. This can be compared to the strategy proposed by Shen and Johnson [77] and Liao et al. [48] of recomputing only the rays that cast modified voxels. The major drawback of Shen and Johnson [77] is that all the voxels of the incremental model must be projected at each frame in order to determine the set of eventually modified rays, even though they actually do not contribute to the image, because of the accumulated opacity. Even more, the projection of these voxels can overlap in the image buffer, which supposes a wasted computational cost.

- Pixel coherency has also been exploited in simultaneous algorithms, that create all a sequence at a time. This idea has not been used in volume data.

- Finally, I/O problems in time-varying scenes have been little addressed for rendering voxel models and much more for isosurface extraction and rendering tetrahedral models. The existing approach [73] for voxel models ray-casting requires

a parallel architecture. There is a lack of solutions for general purpose architectures.

References

[1] K. Akeley. Realityengine graphics. *ACM Computer Graphics*, pages 109–116, 1993.

[2] K. Anagnostou, T. Atherton, and A. Waterfall. 4D volume rendering with the Shear-Warp factorization. *Symp. Volume Visualization and Graphics'00*, pages 129–137, 2000.

[3] J. Arvo and D. Kirk. Fast ray tracing by ray classification. *ACM Computer Graphics*, 21(4):55–64, July 1987.

[4] L. Sobierajski R. Avila and A. Kaufman. Towards a comprehensive volume visualization system. *Volume Visualization 1992*, pages 13–20, October 1994.

[5] S. Badt. Two algorithms for taking advantage of temporal coherence in ray tracing. *The Visual Computer*, 4:55–64, 1988.

[6] C. L. Bajaj, V. Pascucci, G. Rabbiolo, and D. Schikorc. Hypervolume visualization: a challenge in simplicity. In *IEEE Visualization'98*, pages 95–102, 1998.

[7] R. Bakalash and A. Kaufman. The voxel multiple-write bus as an associative processor. *9th Conf. on CAD/CAM & Robotics*, December 1987.

[8] D. Bartz, W. Strasser, O. Gurvit, D. Freudenstein, and M. Skalej. Interactive and multi-modal visualization for neuroendoscopic interventions. *VisSym 2001*, page 9, 2001.

[9] G. Besuievsky and X. Pueyo. Animating radiosity environments through the multi-frame lighting method. *Journal of Visualization and Computer Animation*, pages 93–106, 2001.

[10] A. P. D. Binotto, J. Comba, and C.M. Dal Sasso Freitas. Real-time volume rendering of time-varying data using a fragment-shader compression approach. *6th IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 69–76, 2003.

[11] M. L. Brady, K. K. Jung, H. Nguyen, and T. Nguyen. Interactive volume navigation. *IEEE Trans. on Visualization and Computer Graphics*, 4(3):243–256, 1998.

[12] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping Hardware. *IEEE Volume Visualization'94*, pages 91–98, 1994.

[13] T. W. Calvert J. Chapman and J. Dill. Exploiting temporal coherence in ray tracing. *Graphics Interface*, 1990.

[14] D. Cohen and Z. Shefer. Proximity clouds - an acceleration technique for 3D grid traversal. *Technical Report FC 93-01*, February 1993.

[15] R. Crawfis. Real-time slicing of data space. In *IEEE Visualization'96*, pages 271–277. IEEE Computer Society Press, 1996.

[16] R. Crawfis and N. Max. Texture splats for 3D scalar and vector field visualization. In *IEEE Visualization'93*, pages 261–266, 1993.

[17] B. Csebfalvi, A. Konig, and E. Gröller. Fast maximum intensity projection using binary Shear-Warp factorization. In *WSCG'99*, pages 47–54, 1999.

[18] T. J. Cullip and U. Neumann. Accelerating volume reconstruction with 3D texture mapping hardware. Technical report, University of North Carolina at Chapel Hill, 1993.

[19] F. Dachille, K. Kreeger, B. Chen, I. Bitter, and A. Kaufman. High-quality volume rendering using texture mapping hardware. In *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 69–ff., New York, NY, USA, 1998. ACM.

[20] J. Danskin and P. Hanrahan. Fast algorithms for volume ray-tracing. *1992 Workshop on Volume Visualization*, pages 91–106, 1992.

[21] D. Ellsworth, L. J. Chiang, and H. W. Shen. Accelerating time-varying Hardware volume rendering using TSP trees and color-based error metrics. In *IEEE Visualization'00*, pages 119–128, 2000.

[22] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16, New York, NY, USA, 2001. ACM.

[23] M. Ferré, A. Puig, and D. Tost. A fast hierachical traversal strategy for multimodal visualization. *Visualization and Data Analysis 2004*, pages 1–8, 2004.

[24] J. Freund and K. Sloan. Accelerated volume rendering using homogeneous region encoding. *IEEE Visualization'97*, pages 191–196, 1997.

[25] A. Savopoulos G. Sakas, M. Grimm. Optimized maximum intensity projection (mip). *6th Eurographics Workshop on Rendering*, pages 81–93, June 1995.

[26] A. Van Gelder and K. Kim. Direct volume rendering with shading via 3D textures. In R. Crawfis and C.Hansen, editors, *ACM Symposium on Visualization'96*, pages 23–30, 1996.

[27] A.S. Glassner. Spacetime ray tracing for animation. *IEEE Computer Graphics and Applications*, pages 60–70, March 1988.

[28] E. Grller and W. Purgathofer. Using temporal and spatial coherence for accelerating the calculation of animation sequences. *Eurographics'91*, pages 103–113, 1991.

[29] T. Guenther, C. Poliwoda, C. Reinhard, J. Hesser, R. Maenner, H.-P. Meinzer, and H.-J. Baur. VIRIM: A massively parallel processor for real-time volume visualization in medicine. In *Proc. of the 9th Eurographics Workshop on Graphics HArdware*, pages 103–108, 1994.

[30] H. Hauser, L. Mroz, G. Bischi, and M.E. Gröller. Two-level volume rendering. *IEEE Trans. on Visualization and Computer Graphics*, 7(3):242–252, 2001.

[31] V. Havran, J. Bittner, and H. P. Seidel. Exploiting temporal coherence in ray casted walkthroughs. In *ACM Spring conference on Computer graphics*, pages 149–155, 2003.

[32] V. Havran, C. Damez, and H. P. Myszkowski, K. Seidel. An efficient spatio-temporal architecture for animation rendering. In *Proc. 14th Eurographics workshop on Rendering*, pages 106–117, 2003.

[33] W. Heidrich, M. McCool, and J. Stevens. Interactive maximum projection volume rendering. *IEEE Visualization'95*, pages 11–18, October 1995.

[34] L. Hong, A. Kaufman, Y. C. Wei, A. Viswambharan, M. Wax, and Z. Liang. 3D virtual colonoscopy. *IEEE Computer Graphics & Applications*, pages 26–32, 1995.

[35] J. Huang, K. Mueller, N. Shareef, and R. Crawfis. Fastsplats: optimized splatting on rectilinear grids. In *IEEE Visualization'00*, pages 219–226. IEEE Computer Society Press, 2000.

[36] H. Hubschman and S. W. Zucker. Frame-to-frame coherence and the hidden surface computation: constraints for a convex world. *ACM Trans. Graph.*, 1(2):129–162, 1982.

[37] I. Ihm and R.K. Lee. On enhancing the speed of splatting with indexing. In *IEEE Visualization '95*, pages 69–76. IEEE Computer Society Press, 1995.

[38] A. Pearce K. Sung and C. Wang. Spatial-temporal antialiasing. *IEEE Trans. on Visualization and Computer Graphics*, 2002.

[39] S. Kilthau and T Möller. Splatting optimizations. Technical report, Simon Fraser University, 2001.

[40] T. Klein, M. Strengert, S. Stegmaier, and T. Ertl. Exploiting frame-to-frame coherence for accelerating high-quality volume raycasting on graphics hardware. In *IEEE Visualization '05*, pages 123–230, 2005.

[41] K. Kreeger and A. Kaufman. Mixing translucent polygons with volumes. *Proc. IEEE Visualization*, pages 191–198, 1999.

[42] J. Krüger and R. Westerman. Acceleration techniques for GPU-based volume rendering. In *IEEE Visualization'03*, pages 287–292, 2003.

[43] P. Lacroute and M. Levoy. Fast volume rendering using a Shear-Warp factorization of the viewing transformation. *ACM Computer Graphics*, 28(4):451–458, July 1994.

[44] D. Laur and P. Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *ACM Computer Graphics*, 25(4):285–318, July 1991.

[45] M. Levoy. Efficient ray tracing of volume data. *ACM Trans. on Graphics*, 9(3):245–261, July 1990.

[46] M. Levoy. Volume rendering by adaptive refinement. *The Visual Computer*, 6(1):2–7, 1990.

[47] W. Li, K. Mueller, and A. Kaufman. Empty space skipping and occlusion clipping for texture based volume rendering. In *IEEE Visualization 2003*, pages 317–324, 2003.

[48] S.K. Liao, Y.C. Chung, and J.Z.C. Lai. A two-level differential volume rendering method for time-varying volume data. *The Journal of Winter School in Computer Graphics*, 10(1):287–316, 2002.

[49] W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *ACM Computer Graphics*, 21(4):163–169, July 1987.

[50] E. B. Lum, K. L. Ma, and J. Clyne. A Hardware-assisted scalable solution for interactive volume rendering of time-varying data. *IEEE Trans. on Visualization and Computer Graphics*, 8(3):286–301, 2002.

[51] K. Ma, D. Smith, M. Shih, and H. W. Shen. Efficient encoding and rendering of time-varying volume data. *Technical Report ICASE NASA Langsley Research Center*, pages 1–7, 1998.

[52] E. Maisel and G. Hgron. A realistic image synthesis of animation sequences based on temporal coherence. *Proc. of the Third Eurographics Workshop on Animation and Simulation*, September 1992.

[53] I. Martin, X. Pueyo, and D. Tost. Frame-to-frame coherent animation with two pass radiosity. *IEEE Trans. on Visualization and Computer Graphics*, 9(1):70–84, 2003.

[54] M. Martin, E. Reinhard, P. Shirley, S. Parker, and W. Thompson. Temporally coherent interactive ray tracing. *The Journal of Graphics Tools*, 1(72):41–48, 2003.

[55] Y. Matsushita. Hidden-lines elimination for a rotation object. *Communication of the ACM*, 15(4):245–252, 1972.

[56] D. Meagher. Geometric modeling using octrees encoding. *Computer Graphics and Image Processing*, 19(2):129–147, 1982.

[57] M. Meissner, S. Guthe, and W. Strasser. Interactive lighting models and pre-integration for volume rendering on PC graphics accelerators. In *Proc. of Graphics Interface 2002*, pages 209–218. IEEE Press, 2002.

[58] M. Meissner, U. Hoffmann, and W. Straßer. Enabling classification and shading for 3D texture mapping based volume rendering using OpenGL and extensions. *IEEE Visualization'99*, pages 207–214, 1999.

[59] M. Meissner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis. A practical evaluation of popular volume rendering algorithms. In *IEEE Visualization'2000*, pages 81–91, 2000.

[60] M. Meissner, U. Kanus, and W. Strasser. Vizard ii, a pci-card for real-time volume rendering. In *Proc. of the Eurographics/SIGGRAPH Workshop on Graphics HArdware*, pages 61–68, 1998.

[61] M. Meissner, H. Pfister, R. Westermann, and C. Wittenbrinck. *Volume Visualization and Volume Rendering Techniques*. Eurographics, 2000.

[62] C. Montani, R. Scateni, and R. Scopigno. Discretized marching cubes. *Proc. Visualization'94*, pages 281–287, 1994.

[63] K. Mori, J. Hasegawa, J. Toriwaki, H. Anno, and K. Katada. A fast rendering method using the tree structure of objects in virtualized bronchus endoscope system. *Proc. Visualization in Biomedical Computing*, pages 33–42, September 1996.

[64] L. Mroz and H. Hauser. RTVR: a flexible Java library for interactive volume rendering. In *IEEE Visualization'01*, pages 279–286, 2001.

[65] H. Mueller and R. Crawfis. Eliminating popping artifacts in sheet buffer-based splatting. *IEEE Visualization'98*, pages 239–246, 1998.

[66] K. Mueller, N. Shareef, J. Huang, and R. Crawfis. High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Trans. on Visual. and Computer Graphics*, 5(2):116–134, 1999.

[67] N. Neophytou and K. Mueller. Space-time points: 4D splatting on efficient grids. In *IEEE Symp. on Volume Visualization and graphics*, pages 97–106, 2002.

[68] N. Neophytou and K. Mueller. GPU accelerated image aligned splatting. In I. Fujishiro and E. Gröller, editors, *Volume Graphics*, pages 197–205, 2005.

[69] N. Neophytou, K. Mueller, K. McDonnell, W. Hong, X. Guan, H. Qin, and A.Kaufman. Gpu-accelerated volume splatting with elliptical rbfs. In T. Ertl, K. Joy, and B.Santos, editors, *IEEE-VGTC Symposium on Visualization*, pages 13–20, 2006.

[70] J. Orchard and T. Möller. Accelerated splatting using a 3D adjacency data structure. In *Graphics Interface'01*, pages 191–200, 2001.

[71] H. Pfister and A. Kaufman. Cube-4 - A scalable architecture for real-time volume rendering. *ACM/IEEE Symp. on Volume Visualization*, pages 47–54, October 1996.

[72] A. Puig, D. Tost, and M. I. Navazo. An interactive cerebral blood vessel exploration system. *ACM Visualization'97*, 1997.

[73] E. Reinhard, C.Hansen, and S.Parker. Interactive ray-tracing of time varying data. In *EG Parallel Graphics and Visualisation'02*, pages 77–82, 2002.

[74] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume on standard PC graphics Hardware using multi-textures and multi-stage rasterization. In *HWWS'00: Proc. of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics Hardware*, pages 109–118, New York, NY, USA, 2000. ACM Press.

[75] H. Shen. Visualization of large scale time-varying. *Journal of Physics: Conference series*, 1(46):535–544, 2006.

[76] H. Shen, L. Chiang, and K. Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (tsp) tree. In *IEEE Visualization'99*, pages 371–377, 1999.

[77] H. W. Shen and C. R. Johnson. Differential volume rendering: a fast volume visualization technique for flow animation. In *IEEE Visualization'94*, pages 180–187, 1994.

[78] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In E. Gröller and I. Fujishiro, editors, *Volume Graphics*, pages 187–195, 2005.

[79] K. R. Subramanian and D. F. Fussell. Applying space subdivision techniques to volume rendering. *Proc. Visualization'90*, pages 150–159, 1990.

[80] O. Sudarsky. Exploiting temporal coherence in animation rendering - A survey. *Technical Report CIS9326*, 1993.

[81] T. Theussl, T. Möller, and E. Gröller. Optimal regular volume sampling. *Proc. Visualization'01*, pages 91–98, 2001.

[82] D. Tost. An algorithm of hidden surface removal based on frame-to-frame coherence. *Proc. of Eurographics'91*, pages 261–273, September 1991.

[83] T. van Walsum, A. J. S. Hin, J. Versloot, and F. H. Post. Efficient hybrid rendering of volume data and polygons. In *Proc. of the second EUROGRAPHICS Workshop on Visualization in Scientific Computing*, 1991.

[84] F. Vega, P. Hastreiter, R. Fahlbusch, and G. Greiner. High performance volume splatting for visualization of neurovascular data. In *IEEE Visualization'05*, pages 271–278, 2005.

[85] E. Vergés, S. Grau, and D. Tost. Hardware and software improvements of volume splatting. Technical report, UPC, 2006.

[86] I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek. Interactive global illumination. *Proc. of the 13th Eurographics Workshop on Rendering*, pages 15–24, 2002.

[87] M. Wan, A. Sadiq, and A. Kaufman. Fast and reliable space leaping for interactive volume rendering. In *IEEE Visualization'02*, 2002.

[88] Ming Wan, Qingyu Tang, Arie Kaufman, Zhengrong Liang, and Mark Wax. Volume rendering based interactive navigation within the human colon (case study). In *VIS '99: Proc. of the conference on Visualization '99*, pages 397–400, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[89] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. *Proc. of Visualization 2003*, pages 333–340, 2003.

[90] B. Westermann and T. Ertl. Efficiently using graphics Hardware in volume rendering applications. *ACM SIGGRAPH'98*, pages 169–178, 1998.

[91] R. Westermann. Compression domain rendering of time-resolved volume data. In *IEEE Visualization'95*, page 168. IEEE Computer Society Press, 1995.

[92] L. Westover. Interactive volume rendering. In *Volume Visualization Workshop*, pages 9–16, 1989.

[93] L. Westover. Footprint evaluation for volume rendering. *ACM Computer Graphics*, 24(4):367–376, 1990.

[94] J. Wilhems and A. Van Gelder. A coherent projection approach for direct volume rendering. *ACM Computer Graphics*, 25(4):275–284, July 1991.

[95] J. Wilhems and A. Van Gelder. Multidimensional trees for controlled volume rendering and compression. *ACM Symp. on Volume Visualization*, 11:27–34, October 1994.

[96] J. Woodring, C. Wang, and H. Shen. High dimensional direct rendering of time-varying volumetric data. In *IEEE Visualization'03*, page 55, 2003.

[97] D. Xu and R. Crawfis. Efficient splatting using modern graphics hardware. *Journal of graphics tools*, 8(4):1–21, 2004.

[98] R. Yagel, D. Cohen, and A. Kaufman. Discrete ray tracing. *IEEE Computer Graphics & Applicactions*, 5(12):19–28, 1992.

[99] R. Yagel, D. S. Ebert, J. N. Scott, and Y. Kurzion. Grouping volume renderers for enhanced visualization in computational fluid dynamics. *IEEE Trans. on Visualization and Computer Graphics*, 1(2):117–132, 1995.

[100] R. Yagel and A. Kaufman. Template-based volume viewing. *Eurographics 1992*, 11(3):153–167, 1992.

[101] R. Yagel and Z. Shi. Accelerating volume animation by space-leaping. In *IEEE Visualization'93*, pages 62–69, 1993.

[102] H. Younesy, T. Möller, and H. Carr. Visualization of time-varying volumetric data using differential time-histogram table. In *Volume Graphics'05*, pages 21–29, 2005.

[103] K. J. Zuiderveld, A. H. J. Koning, and M. A. Viergever. Acceleration of ray-casting using 3D distance transforms. *In Proc. of SPIE1808 Visualization in Biomedical Computing*, pages 324–335, 1992.