

Frame-to-frame coherent image-aligned sheet-buffered splatting

S. Grau

D. Tost

May 23, 2006

Abstract

Splatting is a classical volume rendering technique that has recently gained in popularity for the visualisation of point-based surface models. Up to now, there has been few publications on its adaptation to time-varying data. In this report, we propose a novel frame-to-frame coherent view-aligned sheet-buffer splatting of time-varying data, that tries to reduce as much as possible the memory load and the rendering computations taking into account the similarity in the data and in the images at successive instants of time. The results presented in the report are encouraging and show that the proposed technique may be useful to explore data through time.

1 Introduction

In recent years, rendering time-varying volume data has proven to be a powerful tool for the analysis of phenomena such as fluid dynamics, ultrasound motion and cardiovascular dynamics. Typically, there are two main approaches for the exploration of time-varying data: either data are visualised as they are generated [RLGB94], or they are first generated and stored and, next, they are rendered in a post-process. In this report, we focus on the second approach. On one hand, a major problem of this approach is that the large size of the datasets demands huge storage space and excessive amounts of memory to manipulate the data. On the other hand, there is usually a high degree of similarity between data at successive instants of time and even between successive frames of the rendering. This temporal coherency can be conveniently exploited to reduce the amount of data storage and memory load and to speed up rendering computations.

Several techniques have been published that render stored time-varying data: some of them focusing on speeding up isosurfaces extraction for Indirect Volume Rendering (IVR) [WB98] [SCM99] ,[SH00] ,[BWC00] and others focusing at Direct Volume Rendering (DVR). Within this last category, there have been attempts to extend to time-varying data to the four main rendering paradigms: ray-casting [YS93] [SJ94]

[MSSS98] [SCM99] [WSK02] [LCL02] [RCS02] [MRS⁺03], splatting [BPRS98] [NM02], shear-warp [AAW00] and 3D texture mapping [ECS00][LMC02]. However, it is ray-casting that has centred the most attention. Ray-casting is indeed a powerful and flexible volume rendering technique that provides high image quality and benefits from different acceleration strategies. According to [MHB⁺00], splatting can give similar image quality and it may be faster if the occupancy ratio of the model, i.e. the number of relevant voxels in relation to the total number of voxels, is low. In the five last years, the interest for splatting has been renewed because it has proven to be suitable for rendering point-based surfaces [RL00]. Surface splats based on EWA (Elliptical Weighted Average) filtering provide high image quality [ZPvBG02] and they can be implemented using hardware accelerations [CRZP04] [BHZK05] [NM05]. Despite of this interest, the extension of splatting to time-varying data has been little addressed in the bibliography. The main contribution to splatting time-varying data is that of Neophytous and Mueller [NM02] that treat it as a particular case of the general problem of viewing n -dimensional data. In this report, we follow a different approach specifically designed for time-varying data that exploits frame-to-frame coherency.

The report is organised as follows. In section 2, we review previous work on splatting and on time-varying visualisation. In section 3 we present the results of a preliminary study that we have realized in order to estimate the benefits that exploiting frame-to-frame coherence could provide. Based on the conclusions of this study, in section 4 we propose a frame-to-frame coherent splatting algorithm. Finally, in section 7, we present the results of our implementation, previous to the conclusions.

2 Background

2.1 Splatting

The splatting algorithm was proposed by Lee Westover [Wes89]. It considers the volume as an array of overlapping kernels that are projected into the screen plane in order to compose the image. Splatting gains its speed by exploiting the similarity 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.

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*. However, this method may cause colour bleeding and sparkling artifacts because the visibility ordering of the splats is imperfect. To correct this error, Westover [Wes90] proposed the *object-space sheet-buffer splatting* 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 colour 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 [MC98] provided a solution to this problem that also enhances the approximation of the light transport inside voxels: the *image-space sheet-buffer splatting*. 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* [MSHC99], 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.

One of the major advantage of splatting is that only relevant voxels must be splatted and empty and non-selected voxels can be skipped. This idea was first suggested by Yagel et al. [YESK95] 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 [Cra96] 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 colour. Mueller et al. [MSHC99] enhanced the efficiency of the view-aligned sheet-buffer splatting by organising 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. [IL95]. More recently, Orchard and Möller [OM01], 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. Finally, Kiltathau and Möller [KM01] proposed to use run-length encoding (RLE) of the volume in order to skip empty voxels. They construct 24 RLE replications of the volume, which allows them to orderly traverse the volume according to any of the 48 orders. The main drawback to these two last approaches is their storage overhead.

Many efforts have been done in accelerating splatting using hardware. One of the first proposed method [LH91] [WG91] consists of approximating the splat by a collection of polygons, thus taking profit of the hardware-supported polygon rendering pipeline. Crawfis and Max [CM93] 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. [HMSC00] 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 centre. 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 [XC04] proposed two splatting strategies that work on the GPU. The first strategy consists of

using a vertex shader program to generate and render quadrilaterals centred around the voxels centre. 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. [VHFG05] 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 [NM05]. 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. A comparison between different hardware and software-based optimisations of splatting can be found in [VGT06]

2.2 Time-varying direct volume rendering

We here address direct volume rendering. A survey on the use of temporal coherence to speed up for direct and indirect volume rendering can be found in [ACF⁺05]. Previous work on direct time-varying volume rendering typically fall into categories: one that treats separately the time dimension from the spatial dimensions [YS93] [SJ94] [SCM99] [AAW00] [Wes95] [LMC02], and the other (4D rendering) that treats time-varying data as a special case of an n-D model [BPRS98], [ECS00][NM02] [WWS03]. 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. Specifically, various papers address ray-casting a static model from a continuously moving camera. They exploit the *reprojection technique* that computes an image using the previous one, by reprojecting the first visible voxel of each ray according to the new visualisation matrix [GR90]. This technique has been extended to polygonal mesh scenes [AH95]. In both cases, it presents two main drawbacks: first, the presence of holes at pixels onto which no voxels reproject and second, the overlapping of several reprojected points onto one pixel. In order to apply reprojections, Yagel and Shi [YS93] propose to store 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. This strategy works for perspective projections as well as parallel ones, by opposite to that of Gudmundson et al. [GR90], restricted to parallel projections, specifically y-axis rotations. Wan et al. [WSK02] enhance the reprojection by using cells instead of points and using the *Distance To Boundary* (DTB), i.e. the distance of every voxel to the nearest boundary voxel [WTK⁺99]. This strategy increases the memory requirements but it reduces undesirable holes in the reprojected image. Recently, Klein et al. [KSSE05] use these ideas in a GPU-based implementation of ray-casting [KW03] [SSKE05]. They implement the *C-Buffer* as a render target to store the *hit position*, and, 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.

Yoon et al. [YDKN97] uses ray-casting for isosurface rendering. They use a data structure called *image cache* containing a ray-casted image of the volume enriched with 3D information on the visible points. At the next camera position, they project each ray on the image cache and analyse the contents of the pixels of the ray rasterization on the image cache to quickly discard non-intersecting rays and reproject hit positions. For changes in the isovalue of the surface, they store for each ray a piecewise linear approximation of the volume changes along the ray, called *isomap*, that helps them to quickly find the desired isovalue position along the ray.

Shen and Johnson’s ray-casting address time-varying volumes instead of camera changes [SJ94]. They encode differences between consecutive volumes and recast only modified rays. Liao et al. [LCL02] improve this technique by computing an additional differential file that stores the changed pixels positions. The TSP *Temporal Space Tree* [SCM99] 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 [ECS00]. Finally, in a recent paper, [TGFP06], we have proposed a frame-frame coherent strategy based on a double structure: in image-space, a temporal buffer that stores for each pixel the next instant of time in which the pixel must be recomputed, and in object-space a Temporal Run-Length Encoding of the voxel values through time. The algorithm skips empty and unchanged pixels through three different space-leaping strategies. It can compute the images sequentially in time or generate them simultaneously in batch. In addition, it can handle simultaneously several data modalities. Finally, an on-purpose out-of-core strategy is used to handle large datasets.

The temporal extension of the shear-warp technique [LL94] proposed by Anagnostou et al. [AAW00] 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. Finally, Lum et al’s approach [LMC02] 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.

Finally, Ma et al. [MSSS98] propose a technique suitable for ray-casting as well as for splatting. They merge Branch-On-Need Octrees (BONO) [WG94] computed for every instant of time into one structure, the Temporal BONO (T-BONO). This data structure is used together with an auxiliary octree, that stores at each node the partial image corresponding to the subtree. At successive frames, only modified subtrees are visited, and their sub-image is recomputed and composited at the parent level in the hierarchy. The splatting algorithm proposed by Neophitou and Mueller [NM02] belongs to the 4-D rendering approach. They use a 4D Boby Centered Cubic (BCC) grid [TMG01] 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.

3 Preliminary Analysis

We have started our work by investigating the maximum speed up that can be expected if coherence is used to avoid recomputing some of the steps of the image-aligned sheet-buffered splatting pipeline.

3.1 Image-aligned sheet-buffered splatting pipeline

Figure 1 shows the pipeline of the basic image-aligned sheet-buffered splatting. As mentioned in Section 2, the voxel model can be defined as a set of non-disjoint regions, being one of them the empty space. Often, users don't want to see all the volume, but only a subset of its regions. We call *selection*, the specification of this subset. There are different ways of defining a selection. In the original algorithm [MC98], the authors actually define only two regions: the empty one and the selected one defined as one range of property values. Other ways of doing it consist of using auxiliary data structures such as run-length encoding [FPT05] and skeletons [SSC03]. In this report, we have used a set of property ranges. Thus, the input parameters of the pipeline are the voxel model, the selection, and, for shading, the transfer function and eventually a set of lights.

The pipeline is clearly divided in two parts: first, the bucket constructions and filling (BC) and second, the sheets projection and composition steps (SC). As mentioned in Section 2, the first part is done on the CPU whereas, different GPU-based strategies have been proposed for the second part. Changing camera, selection or data values im-

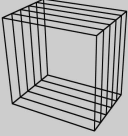
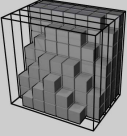
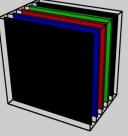
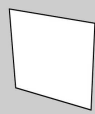

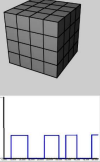
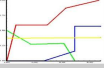
	B C		S C	
	Bucket Construction	Bucket Insertion	Sheet Generation	Sheet Composition
Internal Structures				
Inputs				

Figure 1: Image-aligned sheet-buffered splatting pipeline.

plies recomputing all the pipeline. Changing the transfer function implies recomputing only the second part.

3.2 Analysis of the use of coherence for BC

The brute force implementation of the pipeline depicted in Section 3.1 is to recompute all the BC step at each frame, independently of changes. We call this *Bucket-Construction Brute-Force* (BC-BF) (see Figure 1). Table 1 contains the names of the different algorithms and their acronyms. However, if the camera and the selection do not change, we can avoid recomputing this part of the pipeline. This is the *Bucket-Construction with Bucket Coherence* (BC-BCh) strategy (it uses the same pipeline, see Figure 1). A third approach is to insert all the voxels in the buckets. In this case, the rendering stage processes all the voxels and does not take profit of the natural capability of splatting to perform space-leaping (see Section 2.1). However, the buckets need only to be recomputed if the camera changes, because a change in the selection causes that non-selected property values are assigned to a zero opacity in the transfer function. In this way, we translate the selection step into the second part of the pipeline, that can be done in the GPU. We call this strategy *Bucket-Construction with Bucket Coherence and No Space Leaping* (BC-BCh-NSL) because it uses the coherence of the bucket, i.e. being constant if the camera does not change, and because it does not perform space-leaping (see Figure 2). Finally, a fourth option is to construct an auxiliary array containing the selected voxels. If the selection does not change, the array remains the same. Thus, if the camera moves, it is the array that is traversed instead of the full voxel model to insert the voxels in the buckets. Thus, this strategy exploits array coherence and, as the two first methods, space-leaping. We call it *Bucket-Construction with Array Coherence* (BC-ACh) (see Figure 3).

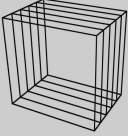
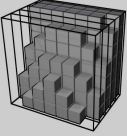
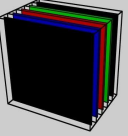
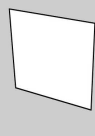


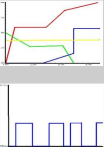
	B C		S C	
	Bucket Construction	Bucket Insertion	Sheet Generation	Sheet Composition
Internal Structures				
Inputs				

Figure 2: BC-BCh-NSL pipeline.


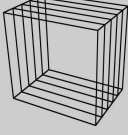
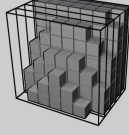
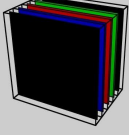
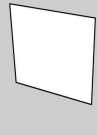
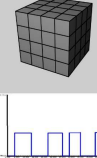



	B C			S C	
	Array Construction	Bucket Construction	Bucket Insertion	Sheet Generation	Sheet Composition
Internal Structures					
Inputs					

Figure 3: BC-ACh pipeline.

3.3 Analysis of the use of coherence for SC

The second part of the pipeline, sheets projection and image composition, needs necessarily to be re-done if the buckets change. However, if they don't change, i.e., if they are composed of the same voxels, the corresponding sheets may be different from the previous instant only if the voxels projected in them change their value. Obviously, exploiting this sheet-to-sheet coherence requires to stores the sheets from instant to instant. However, generally, all buckets change at least in a few voxel values. Thus, previous sheets would hardly be re-usable. Nevertheless, large parts of the sheets remain the same, as for instance, those corresponding to empty space or static parts of the data. As mentioned in Section 2.1, static image-aligned sheet-buffered splatting exploits early splat termination by subdividing the image into tiles and splatting only voxels that project onto tiles that have not reach the maximum opacity. This idea could be extended further by applying the same tile subdivision to the sheets. Therefore, voxels of a bucket that project onto an unchanged tile could be skipped.

Abbreviation	Strategy
BC-BF	Bucket-Construction Brute-Force
BC-BCh	Bucket-Construction with Bucket Coherence
BC-BCh-NSL	Bucket Coherence and No Space Leaping
BC-ACh	Bucket-Construction with Array Coherence
FCh-Alg-Tile	Frame-to-Frame Coherence Splatting Algorithm
BF-Alg	Brute-Force Splatting Algorithm

Table 1: Name and acronym of the different strategies.

In order to assess the expected benefit of skipping voxels projected on unchanged tiles, we have evaluated the number of tiles that remain unchanged in two animation sequences and the expected reduction in the cost of this part of splatting. Again, this is the maximum cost reduction that could be obtained in a frame-to-frame coherent approach of this part of the pipeline, without taking into account the inherent overhead that detecting and processing these buckets and keeping sheets from frame-to-frame.

3.4 Results

We have performed a set of simulations to evaluate the performance of the proposed techniques. All the simulations results shown in the tables are expressed in seconds. They have been measured on a PC Dual Core 3.2 GHz with 3GB of RAM and NVidia 7800 GTX.

We have realized the analysis of BC using static datasets, taking into account only changes in the camera, the transfer function and the set of selected voxels. When data values also vary, the coherence decreases. Therefore, the static case indicates the maximum expectable speed up. For SC, we have used time-varying datasets. The static datasets are (*engine*, *aneurysm*, *mushroom* and *skull*), and the time-varying ones are (*funmushroom* and *five jets*). The datasets come from the repository <http://www.gris.uni-tuebingen.de/areas/scivis/volren/datasets> and <http://www.cs.ucdavis.edu/ma/ITR/tvdr.html>. We have constructed the dataset *funmushroom* by varying the property values of the internal voxels of the *mushroom* dataset. Tables 2 and 3 summarise the characteristics of the datasets. Figures 4 and 5 show a rendered image of the static datasets and a few frames of the time-varying ones.

Dataset	Dimensions	Size	Type	Oc.ratio
Aneurism	256*256*256	16777216	char	0.37%
Skull	256*256*256	16777216	char	5.70%
Engine	256*256*128	8388608	char	20.17%
Mushroom	80*87*59	410640	char	19.22%

Table 2: Characteristics of the static datasets, from left to right: name, size, property value type and occupancy ratio of selected voxels.

Dataset	Dimensions	Size	Frames	Type	Oc.ratio
Five Jets	128*128*128	2097152	40	char	11.64%
FunMushroom	80*87*59	410640	26	char	14.05%

Table 3: Characteristics of the temporal datasets, from left to right: name, size, number of frames, property value type and occupancy ratio of selected voxels.

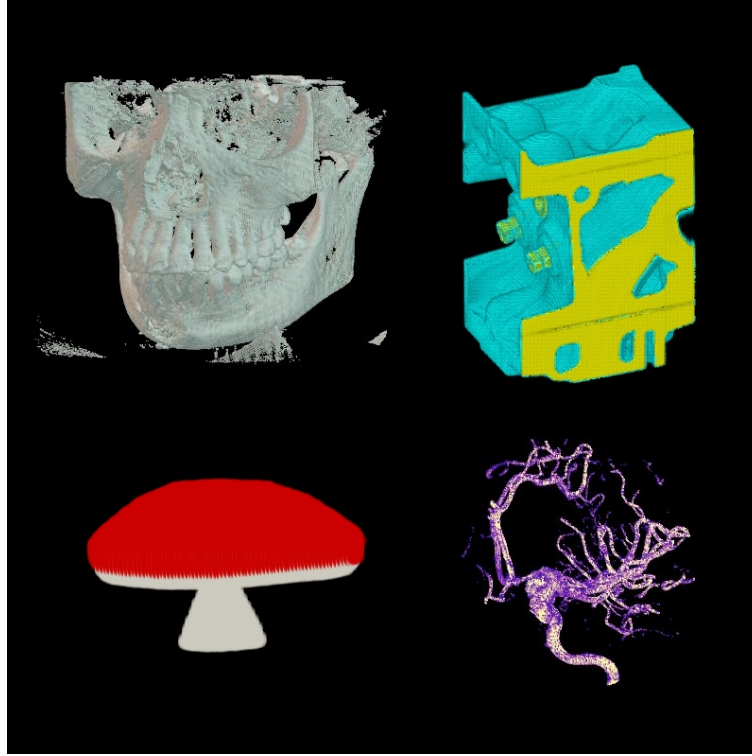


Figure 4: Rendered image of the datasets. Left column from top to bottom: mushroom, engine, aneurysm and skull.

3.4.1 Results for BC

Tables 4, 5 and 6 show the costs in seconds of rendering the different datasets with the four BC methods. The step of projection of the buckets is the same for all four methods except that BC-BCh-NSL projects all the voxels instead of the selected ones. We have separate the costs of each step. Tables 7, 8, 9 and 10 show for each dataset the average cost per frame of the four methods for a 100 frames animation with changes in the camera and the selection.

From these tables we can conclude that taking profit of bucket coherence and array coherence, i.e recomputing the voxel arrays only for changes in the selection and recomputing the buckets only for changes in the camera and selection reduces the rendering cost. Specifically, exploiting the two types of coherence together reduces the

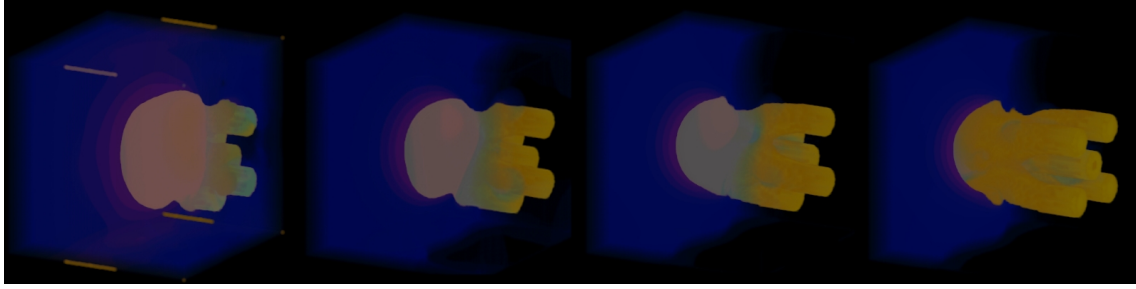


Figure 5: 4 frames of fivejets.

	Engine	Aneurysm	Mushroom	Skull
BC	2.45	3.6	0.13	4.11
SC	8.62	3.24	0.80	7.65
SC (GL)	0.81	0.11	0.06	0.58

Table 4: Costs in seconds of the different steps of the splatting for BC-BF and BC-BCh strategies for the static datasets.

cost up to 24% using GPU for the projection step and up to 76% for the CPU-based projection. Using array coherence supposes a 48% (GPU) and 19% (CPU) of that benefit.

Finally, putting all the voxels in the buckets in order to avoid the selection changes is clearly a bad solution, because the occupancy ratio of the datasets is generally low. Even if the projection step is done on the GPU by opposite to the selection changes step, which is CPU-based, the overall cost in the best case is still 6% (GPU) and 18% (CPU) higher than the basic method.

3.4.2 Results for SC

Tables 11 and 12 show the number of sheet tiles that remain constant in the animation sequence of the *funmushroom* and *five jets* datasets.

These results show that the use of tiles can reduce sheet projection up to 96% on *funmushroom* and 74% on *fivejets*. As expected, the smaller the tiles, the higher the coherency, but the more computationally expensive the tiles management. For

	Engine	Aneurysm	Mushroom	Skull
BC	4.22	9.38	0.20	9.15
SC	12.98	9.10	0.91	12.87
SC (GL)	3.42	6.19	0.16	6.01

Table 5: Costs in seconds of the different steps of the splatting for BC-BCh-NSL strategy for the static datasets.

	Engine	Aneurysm	Mushroom	Skull
Array Creation	2.25	4.8	0.07	4.46
BC	0.62	0.05	0.03	0.46
SC	8.62	3.24	0.80	7.65
SC (GL)	0.81	0.11	0.06	0.58

Table 6: Costs in seconds of the different steps of the splatting for BC-ACh strategy for the static datasets.

Camera changes	Selection changes	BC-BF		BC-BCh		BC-BCh-NSL		BC-ACh	
0	0	11.07	3.26	08.64	0.83	13.02	3.46	08.65	0.84
25	0	11.07	3.26	09.23	1.42	14.04	4.48	08.80	0.99
50	0	11.07	3.26	09.85	2.04	15.09	5.53	08.95	1.14
100	0	11.07	3.26	11.07	3.26	17.20	7.64	09.26	1.45
0	25	11.07	3.26	09.23	1.42	13.02	3.46	09.34	1.53
0	50	11.07	3.26	09.85	2.04	13.02	3.46	10.06	2.25
0	100	11.07	3.26	11.07	3.26	13.02	3.46	11.50	3.69
50	50	11.07	3.26	10.46	2.65	15.09	5.53	10.21	2.40
100	100	11.07	3.26	11.07	3.26	17.20	7.64	11.50	3.69

Table 7: Cost per frame in a 100 frames animation for the engine dataset. Each strategy has the cost in seconds using SC CPU-based and SC GL.

example, in the *funmushroom* we have the same reduction using tiles of 64x64 than 32x32, but we increase the cost of management of all tiles three times.

4 A frame-to-frame coherent splatting

In the previous section, we saw that there exists coherence between consecutive frames in all the steps of the pipeline and that the prediction of this coherence can be useful to reduce the cost of the process. The first part of the pipeline (BC) has these inputs parameters: camera, selection and the dataset. Coherence detection between consecutive frames in this part consists of checking changes in the inputs parameters. To predict any change on camera or selection is not difficult and the consequences are explained in Section 3.2. Any change of a voxel value of the dataset implies that this voxel could modify its selected status. This fact implies that one voxel value change invalidates the current buckets and a new Bucket Insertion is required. The temporal datasets usually change some voxel value every frame. The results of the previous section denote that doing a Bucket Insertion at every frame is very computationally expensive. Two possible strategies can be adopted. First, the Bucket Insertion can be done incrementally by adding at each frame the new selected voxels and removing from them those that are not selected at that frame. An alternative, is to insert in the buckets all the voxels whose value is selected at some of the frames. In that second case, the opacity of vox-

Camera changes	Selection changes	BC-BF		BC-BCh		BC-BCh-NSL		BC-ACh	
0	0	6.84	3.71	3.28	0.15	09.19	06.28	3.28	0.15
25	0	6.84	3.71	4.14	1.01	11.45	08.54	3.30	0.17
50	0	6.84	3.71	5.04	1.91	13.79	10.88	3.31	0.18
100	0	6.84	3.71	6.84	3.71	18.48	15.57	3.33	0.20
0	25	6.84	3.71	4.14	1.01	09.19	06.28	4.36	1.23
0	50	6.84	3.71	5.04	1.91	09.19	06.28	5.47	2.34
0	100	6.84	3.71	6.84	3.71	09.19	06.28	7.70	4.57
50	50	6.84	3.71	5.94	2.81	13.79	10.88	5.48	2.35
100	100	6.84	3.71	6.84	3.71	18.48	15.57	7.70	4.57

Table 8: Cost per frame in a 100 frames animation for the aneurysm dataset. Each strategy has the cost in seconds using SC CPU-based and SC GL.

Camera changes	Selection changes	BC-BF		BC-BCh		BC-BCh-NSL		BC-ACh	
0	0	0.93	0.19	0.80	0.06	0.91	0.16	0.80	0.06
25	0	0.93	0.19	0.83	0.09	0.96	0.21	0.81	0.07
50	0	0.93	0.19	0.87	0.13	1.01	0.26	0.82	0.08
100	0	0.93	0.19	0.93	0.19	1.11	0.36	0.83	0.09
0	25	0.93	0.19	0.83	0.09	0.91	0.16	0.83	0.08
0	50	0.93	0.19	0.87	0.13	0.91	0.16	0.85	0.11
0	100	0.93	0.19	0.93	0.19	0.91	0.16	0.90	0.16
50	50	0.93	0.19	0.90	0.16	1.01	0.26	0.86	0.12
100	100	0.93	0.19	0.93	0.19	1.11	0.36	0.90	0.16

Table 9: Cost per frame in a 100 frames animation for the mushroom dataset. Each strategy has the cost in seconds using SC CPU-based and SC GL.

els of the bucket that are not selected at the current frame should always be set to zero. In this report, we have implemented the second strategy.

Exploiting the frame-to-frame coherence in the second part (SC) implies to detect unchanged sheet tiles and to avoid their projection. A tile remains constant while of all its selected voxel values remain constant. Therefore, the next instant of change of a tile corresponds to the first instant in which there is a change in the selected status of one of its voxels or a change of value of one selected voxel. In order to implement this idea, we encode the time-varying volume as a Temporal Run-Length. This encoding allows us to quickly compute at any time the next instant of change of any voxel. Using this new structure in the Sheet Projection we can compute how many frames this tile will remain constant if there is no change in the previous BC steps or in the SC inputs parameters (transfer function or lighting in surface shading).

Camera changes	Selection changes	BC-BF		BC-BCh		BC-BCh-NSL		BC-ACh	
0	0	11.76	4.69	07.69	0.62	12.96	06.10	07.70	0.63
25	0	11.76	4.69	08.68	1.61	15.16	08.30	07.81	0.74
50	0	11.76	4.69	09.71	2.64	17.45	10.59	07.92	0.85
100	0	11.76	4.69	11.76	4.69	22.02	15.16	08.15	1.08
0	25	11.76	4.69	08.68	1.61	12.96	06.10	08.88	1.81
0	50	11.76	4.69	09.71	2.64	12.96	06.10	10.11	3.04
0	100	11.76	4.69	11.76	4.69	12.96	06.10	12.57	5.50
50	50	11.76	4.69	10.73	3.66	17.45	10.59	10.23	3.16
100	100	11.76	4.69	11.76	4.69	22.02	15.16	12.57	5.50

Table 10: Cost per frame in a 100 frames animation for the skull dataset. Each strategy has the cost in seconds using SC CPU-based and SC GL.

	Tiles 256x256	Tiles 128x128	Tiles 64x64	Tiles 32x32
number tiles	9256	37024	148096	592384
recomputed tiles	2555	4296	10156	32609
ratio	0.18	0.08	0.04	0.04

Table 11: Analysis of SC for *funmushroom*. This example has 89 buckets.

5 Data structures

5.1 Buckets

In static algorithms, the buckets only store their voxels. We subdivide each bucket into tiles, and each tile stores those voxels that project on it. We also store the sheet tiles and the number of frames that they remain constant.

5.2 Temporal Run-Length

The *Temporal Run-Length* (TRL) representation stores for every voxel v_i a sequence of codes composed of the voxel value and the next frame at which this value changes:

$$codes(v_i) = \langle value_k, tnext_k \rangle, k = 1 \dots ncodes(v_i).$$

The query for the value of a voxel v_i at a given frame t requires, with this structure,

	Tiles 256x256	Tiles 128x128	Tiles 64x64	Tiles 32x32
number tiles	21120	84480	337920	1351680
recomputed tiles	17744	31957	95990	351437
ratio	0.84	0.38	0.28	0.26

Table 12: Analysis of SC for *fivejets*. This example has 132 buckets.

a search in $codes(v_i)$ of the code whose time span contains frame t . In order to avoid this search and access directly to the searched code, we add to the structure a pointer that is set to the first code at the beginning of the sequence and that is updated to the current code during the traversal. Therefore, assuming that a simple byte is sufficient to store the number of frames of the codes and that the pointer to the current code is also a byte, the occupancy in bytes of the TRL structure for a modality m occupying nb_m bytes per voxel of a voxel model composed of nv_m voxels is:

$$Occup(TRL_m) = \sum_{i=1}^{nv_m} (1 + ncodes(v_i) * (nb_m + 1))$$

This occupancy can be compared to the occupancy of the regular voxel model along time:

$$Occup(VoxelModel_m) = nv_m * nf_m * nb_m,$$

being nf_m the number of frames of modality m .

We call r_{occ} the ratio between the occupancy of the TRL and the regular voxel model:

$$r_{occ} = \frac{Occup(TRL)}{Occup(VoxelModel)}.$$

This ratio has a direct relationship with the temporal coherency of the voxel model, since it depends on the number of voxels that change. As it is obvious, the TRL cannot be constructed on static models, composed of one frame, because it would triplicate their occupancy. In the worst case, for an animated model all the voxels change at every frame and, thus, the ratio of occupancy is:

$$r_{occ} = 1 + \frac{nb_m + nf_m + 1}{nb_m * nf_m}$$

more than 2 when the bytes per property of modality m is $nb_m = 1$. However, this is less than the worst case of the incremental model proposed in [SJ94] which can be four times more the original one.

Nevertheless, if the temporal coherency is high, this ratio can be very small. In these cases, the TRL is a compressed representation of the temporal evolution of the model.

The TRL is computed in a pre-process that first loads the voxel model corresponding to the first frame and initialises the list of codes for every voxel. Next, the voxel models at the following frames are loaded one-by-one and traversed. For every voxel, the value of the current code in the TRL is compared to the value of the loaded voxel model. If the two values are equal, the frame of the current code is updated, otherwise a new code is constructed. Variations of the property values of empty voxels are not considered for the creation of new codes. Therefore, if a voxel has a variable value, but empty through all the sequence, it has a unique code. This pre-processing has a cost complexity $Cost_{pp} = O(nv_m * nf_m)$.

6 Algorithm

As mentioned in Section 4, the proposed frame-to-frame coherent strategy tries to skip all the steps that remains unchanged from previous instant. The Bucket Creation is done only if the camera changes. The Array Construction is recomputed in case that selection is modified. The execution of any of the two previous steps implies a new Bucket Insertion. Before the SC part, the transfer function is modified in order to assign to zero the opacity of the values non selected at the current frame. Finally, this strategy traverses each tile of a bucket and projects it only when it needs to be recomputed or a change in a previous step has happened. Algorithm 1 shows all the process.

Algorithm 1 Frame-to-frame Coherent Algorithm

```
for all frame f do
    bucket_insertion = false
    if CameraChange() then
        CreateBuckets()
        bucket_insertion = true
    end if
    if SelectionChange() then
        CreateArray()
        bucket_insertion = true
    end if
    if bucket_insertion then
        InsertVoxelsBuckets()
        tile_projection = true
    else
        tile_projection = false
    end if
    ModifyTransferFunction()
    for all bucket b do
        SheetInit(b)
        for all tile t of b do
            if TileChange(t) or tile_projection then
                ProjectTileSheet(t)
            end if
            ComposeTileSheet(t,b)
        end for
    end for
end for
```

7 Results

We have performed a set of simulations to evaluate the performance of the proposed techniques. The simulations are done with the same hardware as in Section 3.4 and using the same temporal datasets described there (*fivejets* and *funmushroom*).

We call the coherent algorithm *FCh-Alg-Tile*, where *Tile* is the size of the tiles used, i.e. *BCh-Alg-128x128*. In order to evaluate the benefits of the proposed strategy, we have compared it with a brute-force strategy that we have called *BF-Alg*. This brute force approach does not use the voxel array, it performs the Bucket Insertion of the current selected voxels at each frame, and it does not use sheet tiles.

Algorithm 2 BC-BCh Algorithm

```

for all frame f do
  if CameraChange() then
    CreateBuckets()
  end if
  InsertSelectedVoxelsBuckets()
  for all bucket b do
    SheetInit(b)
    ProjectSheet(t)
    ComposeSheet(t,b)
  end for
end for

```

Tables 13 and 14 show the costs in seconds of rendering the different steps of the pipeline for each algorithm. We have measured the case best and worst case costs for CPU-based and GPU-based SC step. The best case implies that there is no change in selection and camera, and consequently the tiles that remain constant are not projected. The worst case is when all the tiles require to be projected.

	Array Construcion	Bucket Insertion	SC		SC GL	
BF-Alg	0	0.08	0.89		0.05	
FCh-Alg-256x256	1.85	0.14	0.83	1.08	0.05	0.11
FCh-Alg-128x128	1.85	0.47	0.79	1.11	0.06	0.20
FCh-Alg-64x64	1.85	1.82	0.71	1.14	0.08	0.60

Table 13: Cost per frame in seconds for the *funmushroom* dataset animation. Steps of SC for the proposed strategy has the value of the using the coherence of the model for the unchanged tiles and the recomputing all the tiles every frame.

The results for *funmushroom* dataset show that in the best case the cost of SC is reduced in 20%, but in the worst case the overhead cost of management increases the cost of SC in 28%. However, in the simulations, we have not applied early termination. Therefore, all sheets are processed and divided into tiles. This can be simplified and

many of this tiles can become useless if early termination is applied. With the GPU-based SC strategy, the results are not satisfactory in any case because the management of the coherence is done in the CPU. Therefore, the goal is to perform the proposed strategy entirely in the GPU.

	Array Construcion	Bucket Insertion	SC		SC GL	
BF-Alg	0	0.39	1.29		0.12	
FCh-Alg-256x256	4.94	3.35	1.89	2.08	0.75	0.94
FCh-Alg-128x128	4.94	11.47	1.73	2.00	0.79	1.03
FCh-Alg-64x64	4.94	43.50	1.60	1.94	0.95	1.05

Table 14: Cost per frame in seconds for the *fivejets* dataset animation. Steps of SC for the proposed strategy has the value of the using the coherence of the model for the unchanged tiles and the recomputing all the tiles every frame.

The results for *fivejets* dataset show the significance of the coherence of the dataset. The *funmushroom* dataset has a high coherence, because the selected voxels have a selected value during a large part of instants. The *fivejets* dataset are less coherent. Therefore, a large number of voxels are on the buckets but not much contribute to the projection because they have opacity zero. The proposed algorithm cannot skip these voxels because they are taken into account for computing the tile duration. This problem can be avoided by performing an incremental Bucket Insertion step. This is the next step of our research.

8 Conclusions

In this report, we have analysed the possibility of exploiting frame-to-frame coherence in image-aligned sheet-buffered splatting. First, we have performed a set of empirical tests to determine the weight of each step of the algorithm in the overall cost and how this cost is affected by the variation of input parameters of the rendering pipeline (camera, transfer function and selection) and by the variation of data values through time. This analysis has shown that exploiting temporal coherence could lead to high reductions of the computational cost and therefore it is worth to investigate means of taking profit of this coherence.

In the second part of report, we have proposed a first version of a frame-to-frame coherent splatting algorithm. The results are encouraging but, the method presented has a high overhead cost and it needs to be improved. Consequently, we will continue our research in three directions. First, we will investigate how to compute the buckets incrementally instead of inserting in them at the beginning of the sequence all the voxels that can be useful at some instant. Next, we will perform more tests using early splat termination and trying to avoid the management of all the tiles. Furthermore, we will investigate how to group consecutive buckets that use shared tiles in order to

reduce the high number of tiles stored and computed. Finally, once the method will be tuned in software, we plan to design a strategy to perform all the process in the GPU.

A step further in our research is to extend this strategy to large volumes of time-varying data by designing out-of-core approaches to reduce the memory management overhead.

References

- [AAW00] K. Anagnostou, T. Atherton, and A. Waterfall. 4D volume rendering with the Shear-Warp factorization. *Proc. Symp. Volume Visualization and Graphics'00*, pages 129–137, 2000.
- [ACF⁺05] D. Ayala, J. Campos, M. Ferré, S. Grau, A. Puig, and D. Tost. Research report lsi-05-30-r: Time varying volume visualization. Technical report, Dep. LSI, UPC, 2005.
- [AH95] S. J. Adelson and L. F. Hodges. Generating exact ray-traced animation frames by reprojection. *IEEE Computer Graphics and Applications*, 15(3):43–52, 1995.
- [BHZK05] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High quality surface splatting on today's GPU. In M. Pauly and M. Zwicker, editors, *EG Symp. on Point-based graphics*, pages 1–8, 2005.
- [BPRS98] C. L. Bajaj, V. Pascucci, G. Rabbio, and D. Schikore. Hypervolume visualization: a challenge in simplicity. In *IEEE Visualization'98*, pages 95–102, 1998.
- [BWC00] P. Bhaniramka, R. Wenger, and R. Crawfis. Isosurfacing in higher dimensions. In *IEEE Visualization'00*, pages 267–273, 2000.
- [CM93] R. Crawfis and N. Max. Texture splats for 3D scalar and vector field visualization. In *IEEE Visualization'93*, pages 261–266, 1993.
- [Cra96] R. Crawfis. Real-time slicing of data space. In *IEEE Visualization'96*, pages 271–277. IEEE Computer Society Press, 1996.
- [CRZP04] W. Chen, L. Ren, M. Zwicker, and H. Pfister. Hardware-accelerated adaptive EWA volume splatting. In *IEEE Visualization'04*, pages 67–74, 2004.
- [ECS00] 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 symposium on Volume visualization*, pages 119–128, 2000.

- [FPT05] M. Ferré, A. Puig, and D. Tost. Decision trees for accelerating unimodal, hybrid and multimodal rendering models. *The Visual Computer*, pages 305–313, 2005.
- [GR90] B. Gudmundsson and M. Randén. Incremental generation of projections of ct volumes. In *Proceedings of the first conference on Visualization on biomedical computing 1990*, pages 27–34, 1990.
- [HMSC00] 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.
- [IL95] 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.
- [KM01] S. Kiltathau and T Möller. Splatting optimizations. Technical report, Simon Fraser University, 2001.
- [KSSE05] T. Klein, M. Strengert, S. Stegmaier, and T. Ertl. Exploiting frame-to-frame coherence for accelerating high-quality volume raycasting on graphics hardware. In *Vis05: Proceedings of the conference on Visualization ’05*, pages 123–230. IEEE Press, 2005.
- [KW03] J. Krüger and R. Westerman. Acceleration techniques for GPU-based volume rendering. In *IEEE Visualization’03*, pages 287–292, 2003.
- [LCL02] 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.
- [LH91] D. Laur and P. Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *ACM Computer Graphics*, 25(4):285–318, July 1991.
- [LL94] 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.
- [LMC02] 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.
- [MC98] H. Mueller and R. Crawfis. Eliminating popping artifacts in sheet buffer-based splatting. *IEEE Visualization’98*, pages 239–246, 1998.

- [MHB⁺00] 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.
- [MRS⁺03] 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.
- [MSHC99] K. Mueller, N. Shareef, J. Huang, and R. Crawfis. High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Trans. on Visualization and Computer Graphics*, 5(2):116–134, 1999.
- [MSSS98] K. Ma, D. Smith, M. Shih, and H.W. Shen. Efficient encoding and rendering of time-varying volume data. *Technical Report ICASE NASA Langley Research Center*, pages 1–7, 1998.
- [NM02] 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.
- [NM05] N. Neophytou and K. Mueller. GPU accelerated image aligned splatting. In I. Fujishiro and E. Gröller, editors, *Volume Graphics*, pages 197–205, 2005.
- [OM01] J. Orchard and T. Möller. Accelerated splatting using a 3D adjacency data structure. In *Graphics Interface'01*, pages 191–200, 2001.
- [RCS02] 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.
- [RL00] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Siggraph'2000*, pages 343–352, 2000.
- [RLGB94] J. Rowlan, G. Lent, N. Gokhale, and S. Bradshaw. A distributed, parallel, interactive volume rendering package. In *IEEE Visualization'94*, pages 21–30, 1994.
- [SCM99] 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.
- [SH00] P. Sutton and C. Hansen. Accelerated isosurface extraction in time-varying field. *IEEE Trans. on Visualization and Computer Graphics*, 6(2):98–107, 2000.

- [SJ94] 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.
- [SSC03] V. Singh, D. Silver, and N. Cornea. Real-time volume manipulation. In *Volume Graphics*, pages 45–52, 2003.
- [SSKE05] 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.
- [TGFP06] D. Tost, S. Grau, M. Ferré, and A. Puig. Ray-casting time-varying volume data sets with frame-to-frame coherence. *IEEE/SPIE International Conference on Visualization and Data Analysis 2006*, 2006.
- [TMG01] T. Theussl, T. Möller, and E. Gröller. Optimal regular volume sampling. *Proc. Visualization'01*, pages 91–98, 2001.
- [VGT06] E. Vergés, S. Grau, and D. Tost. Hardware and software improvements of volume splatting. Technical report, UPC, 2006.
- [VHFG05] 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. IEEE Computer Society Press, 2005.
- [WB98] C. Weigle and D.C. Banks. Extracting iso-valued features in 4-dimensional scalar fields. In *Symp. on Volume Visualization*, pages 103–110, 1998.
- [Wes89] L. Westover. Interactive volume rendering. In *Chapel Hill Volume Visualization Workshop*, pages 9–16, 1989.
- [Wes90] L. Westover. Footprint evaluation for volume rendering. *ACM Computer Graphics*, 24(4):367–376, July 1990.
- [Wes95] R. Westermann. Compression domain rendering of time-resolved volume data. In *IEEE Visualization'95*, page 168. IEEE Computer Society Press, 1995.
- [WG91] J. Wilhems and A. Van Gelder. A coherent projection approach for direct volume rendering. *ACM Computer Graphics*, 25(4):275–284, July 1991.
- [WG94] J. Wilhems and A. Van Gelder. Multidimensional trees for controlled volume rendering and compression. *Proc ACM Symp. on Volume Visualization*, 11:27–34, October 1994.

- [WSK02] M. Wan, A. Sadiq, and A. Kaufman. Fast and reliable space leaping for interactive volume rendering. In *IEEE Visualization'02*, 2002.
- [WTK⁺99] 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: Proceedings of the conference on Visualization '99*, pages 397–400, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
- [WWS03] J. Woodring, C. Wang, and H-W Shen. High-dimensional direct rendering of time-varying volumetric data. In *IEEE Visualization 2003*, pages 417–424, 2003.
- [XC04] D. Xu and R. Crawfis. Efficient splatting using modern graphics hardware. *Journal of graphics tools*, 8(4):1–21, 2004.
- [YDKN97] I. Yoon, J. Demers, T. Kim, and U. Neumann. Accelerating volume visualization by exploiting temporal coherence. In *Vis97: Proceedings of the conference on Visualization'97*, pages 21–24. IEEE Press, 1997.
- [YESK95] 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.
- [YS93] R. Yagel and Z. Shi. Accelerating volume animation by space-leaping. In *IEEE Visualization'93*, pages 62–69, 1993.
- [ZPvBG02] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. EWA splatting. *IEEE Trans. on Visualization and Computer Graphics*, 8(3):223–238, 2002.