A comparative analysis of hardware and software improvements of volume splatting

E. Vergés , S. Grau and D. Tost

May 18, 2006

Abstract

This paper compares different hardware-based accelerations of the three classical splatting strategies: *composite-every-sample*, *object-space sheet-buffer* and *image-space sheet-buffer*. Specifically, we analyze the use of point sprites and 2D textures for the splat's projection, the use of frame-buffer objects for the buffer composition and the use of GPGPU techniques for the transformation of voxels. We compare the efficiency of the hardware-accelerated splatting with 3D-texture mapping. In addition, we compare the rendering speed-up provided by hardware accelerations with two software-based space-leaping techniques: run-length encoding of labeled voxel models and voxel arrays.

1 INTRODUCTION

The development of the last generations of graphics hardware has dramatically changed computer graphics research. Scientific visualization have gained many benefits from these advances, because they provide significant accelerations of rendering. Besides, in visualization, the ever-growing size of the datasets yields to a permanent demand of more computing power as well as hardware-based and software-based optimizations. A good example of the influence of graphics hardware technology in visualization is the current popularity of 3D-texture slicing for volume rendering. This technique consists basically of loading the entire volume in video memory and resampling it onto parallel proxy-geometry planes that are composed back-to-front to create the final image. Its speed is due to its use of the embedded trilinear interpolation hardware. Since this idea was first proposed in 1993 [CN93], the 3D-texture slicing technique has evolved and now, it incorporates volume shading [GK96], surface shading [WE98] and interactive classification [MHS99]. This evolution has been done in parallel and thanks to the graphics hardware improvements.

The classical volume ray-casting can also be improved by a clever use of hardware capabilities. As an example, Mitsubishi's VolumePro is a special purpose ray-casting implementation [PHK*99]. UltraVis [Kni00] is an assembler ray-casting implementation for specific types of CPUs. Finally, Kruger and Wester-

man [KW03] have proposed a multipass ray-casting with early ray-termination that takes profit from 3D-textures, z-test and fragment shaders.

In the last five years, splatting, originally designed to render volume datasets, has gained popularity because its use has been extended to render point-based surface representations [PZBG00]. Splatting considers the volume or the surface as an array of overlapping kernels that are projected onto the screen plane in order to compose the image. Many attempts have been done to accelerate this technique using the GPU capabilities for both surface [RPZ02] and volume splatting [XC04]. Most of the research in this area focuses on the acceleration of the kernels projection using texture maps [CM93], 1D and 2D look-up tables *(fastSplats)* [HMSC00] and point sprites [BK03]. The acceleration of other stages of the splatting pipeline such as the use of hardware-assisted opacity convolution [MSHC99], plane composition [NM05] and voxels geometrical transformation have been less addressed and restricted to *image-space sheet-buffer splatting*. Moreover, several software-based accelerations of splatting have been proposed, such as run-length encoding [KM01] and computation of render lists [MH01]. These techniques are primarily aimed at reducing the number of processed voxels. They have proved to greatly reduce the computational cost of splatting. However, the acceleration provided by these techniques in comparison with that provided by hardware is not very well known. In this paper, we address splatting of volume models. We analyze and compare different hardware-driven splatting accelerations for three different voxel model traversal schemes. We implement them and compare the results with software accelerations and 3D-texture mapping.

2 PREVIOUS WORK

The splatting algorithm was proposed by Lee Westover [Wes89]. This algorithm 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. 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 color bleeding and sparkling artifacts because the visibility ordering of 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 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 [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. This algorithm is composed of two main steps. In the first one, the voxel model is traversed, the voxels to be rendered are first transformed according to the viewing matrix, then depth-sorted and finally inserted into *buckets* such that each bucket covers the distance between suceesive sheet buffers. In the second stage, the sheets are processed in Front-to-Back (FTB) order. For each voxel in a sheet, the proper footprint is chosen according to a fast indexing scheme. In the *image-space sheet-buffer splatting* [MSHC99], early splat elimination is possible in FTB composition 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 advantages 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 coordinates in the plane and their values. 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 color. A similar idea is exploited in the RTVR system [MH01] that uses an intermediate array of slice-sorted *RenderLists* for each structure of the volume that store the voxels of the slices that are relevant for rendering. This structure is also used in the *Two-level rendering* proposed by Hauser et al. [HMBG01] [HBH03]. Mueller at al. [MSHC99] enhanced the efficiency of the view-aligned sheet-buffer splatting by organizing the selected voxels into *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 at al. [IL95]. More recently, Orchard and Möller [OM01], proposed to use a list of adjacency data structure in which each non-empty voxel in a scan list is linked to the next non empty voxel in the scan-line. Finally, Kilthau and Möller [KM01] proposed to use run-length encoding (RLE) 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 methods [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 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 [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 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 PBuffer with additive blending. Next, the GL convolution flag is activated and a texture is copied from the PBuffer using *glTexSubImage2D* such that each texel is a convolution between the PBuffer 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, i.e images that do not provide depth clues, because opacity is simply accumulated without alpha-blending. 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.

3 HARDWARE ACCELERATIONS

We have developed hardware-based accelerations for the three main splatting strategies: Composite-Every-Sample (CES), Object-space Sheet-buffer Splatting (OSS) and Image-space Sheet-buffer Splatting (ISS). Six main operations can be identified in the rendering pipelines of these three strategies:

- 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)

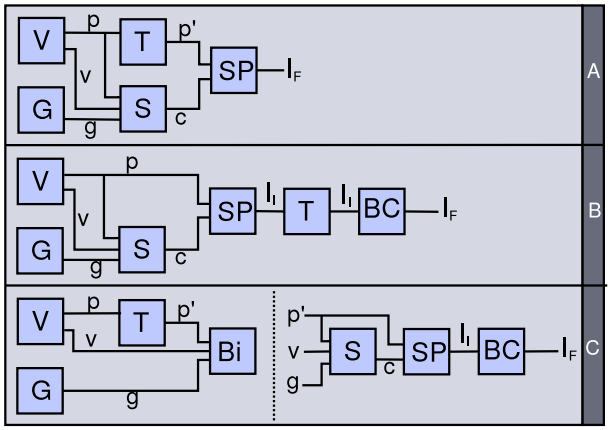

Figure 1: Rendering pipelines of the three splatting strategies from top to down: Composite-Every-Sample (CES), Object-space Sheet-buffer Splatting (OSS) and Image-space sheet-buffer Splatting (ISS). p is the point in world coordinates and p' in viewing coordinates, g is the gradient, v the voxel value, c the resulting color and I_i the i-th bucket and I_F the frame-buffer image.

8cm

The rendering pipelines that we have implemented for these three strategies are depicted in Figure 1. The ISS pipeline is split into two-steps: the creation of the buckets and the splatting itself. In the second step, the point's coordinates and color are fetched directly from their associated buckets. Observe that in order to get comparable results, the ISS pipeline that we have implemented is the classical one that shades in object-space and not the one suggested by [NM05], that computes pixel gradients and applies shading in the sheet-buffers. The 3D texture-mapping that we have used in the comparisons uses pre-shaded RGBα texture loaded into the GPU memory. For splatting, the voxel access and gradient computations are done in the CPU. We have designed and tested hardware-based accelerations of the geometrical transformations (T), shading (S), splatting (SP) and buffer-composition (BC).

3.1 Splats projection

Software-based splatting consists of rasterizing one point per voxel and compositing all the pixels of the footprint in the image buffer or the sheet-buffer around the rasterized point. We have analyzed two hardware-based splatting alternatives. First, the splat can be loaded as a 2D texture and a four vertices textured *quad* can be projected. If the splat is defined in object-space, the texture needs only to be loaded once, but the computation of the *quad* must be done in object space, which is computationally expensive. Alternatively, the splat can be computed in image-space, for each camera position and, thus, the textured *quad* can be defined directly in image space setting a 2D ortho

camera before its projection. An intermediate possibility between these two is to precompute the splat textures for a set of pre-defined cameras. However, this increases considerably the memory requirements. It is not even feasible in the case of ISS that requires at least eight footprints per camera. The second hardware based strategy is to use the *glPointSprite*.

3.2 Buffers composition

One of the bottlenecks of the OSS and ISS strategies is the sheets composition with α-blending into the image buffer. We have substituted this step by using the Frame Buffer Objects extension (FBO). We splat the voxels using FBOs as offscreen render targets. Next, we activate α-blending; we define a 2D *quad* of the size of the frame-buffer; we associate to it the texture of the FBO and we render it in the frame-buffer. In this way, the sheet is composed with the frame-buffer using hardware acceleration.

3.3 Shading

We have implemented three shading of increasing computational cost:

- Per-value shading that simply uses the property value as a gray or RGB color RGB color
- Volume shading that gets the emission and opacity from a Look-Up-Table (LUT) implementation of the transfer function
- Surface shading that gets a material identifier from LUTs, and then applies the Phong model.

Per-value shading is the simplest one. If voxel property values are stored as 1 byte (gray) or 3 bytes (RGB), it requires no computations. In order to accelerate volume shading, we have implemented the LUT as a 1D texture. The voxel property value is used as an index to this texture. In order to combine this shading with the texture-based and point-sprite splat projection, a multitexturing context must be set. The 1D texture is activated first and its corresponding color is blended to all the pixels of the 2D texture. Surface shading can also be handled using a hardware-driven LUT, for a pre-defined set of scalar product between the lighting direction and the gradient vectors. The computational cost of surface shading is, in this case, the same as volume shading without taking into account the table construction. Alternatively, the voxel property value is used as an index to a software-based materials LUT. The optical properties of the surface are set using *glMaterial*, the gradient vector is used to set *glNormal* and, if OpenGL lighting is on, Phong's model is applied. The main drawback of this approach is that, even if only the surface color is set for each voxel, considering constant the specular color and exponent, it requires, for each surface voxel, at least either one *glMaterial* instruction or one *glColorMaterial* causing the material color to track the current color. per surface voxel. Other shading

models than Phong could be applied using vertex shaders, but we believe that this would not further enhance the speed of shading.

3.4 Viewing transformations

In CES, the viewing transformations can be done using OpenGL hardware-supported matrix operations if voxels are treated as *quad* or *glPointSprite*, as described in Section 3.1. In OSS, the viewing transformation is handled during the rendering of the *Frame Buffer Objects* (FBO) into the frame-buffers. However, in ISS, the viewing transformation is done in the CPU during the creation of the buckets. As an alternative, we have implemented a General Purpose Computation on the GPU (GPGPU) to load the selected voxels into the GPU as a floating point texture, transform them into the viewing coordinate system and generate a transformed list of voxels that can be used to insert each voxel in the proper bucket.

4 SOFTWARE ACCELERATION

We have implemented two software accelerations to cull non-selected voxels. In the first strategy, we first traverse the full voxel model and compute a run-length encoding of the classification values of the model, such that each code is composed of a material identifier and the number of consecutive voxels sharing this material. During rendering, users select a set of materials for rendering; the run-length encoding is traversed but only the voxels corresponding to selected codes are processed. In order to allow any camera position, three run-length encoding are computed, one for each of the coordinate planes. Therefore,this model is suitable for OSS splatting or to create the buckets in ISS. It could be extended to CES, but with the drawback that 26 codifications should be computed in this case.

The second strategy is used to accelerate ISS. Once the selection is done, the selected voxels are stored in a array that keeps the voxel coordinates and their value. For each camera position, instead of traversing all the voxels, only those of the array are processed.

5 RESULTS

We have implemented and tested the different algorithms in our software platform *Hipo*. Table 1 indicates the abbreviations used for the seven different algorithms analyzed.

To test the algorithms we have used six datasets that differ in various aspects: their size, that varies from the smallest 64^3 digitized dinosaur (*dino*) to the larger micro-CT scanned rabbit femur (*femur*); their property value type 1 byte o 4 bytes (RGB α); their occupancy ratio measured as the number of

Abbreviation	Algorithm
CES	Composite-Every-Sample
H-CES	Hardware-based Composite-Every-Sample
OSS	Object-space Sheet-buffer
H-OSS	Hardware-based Object-space Sheet-buffer
ISS	Image-space Sheet-buffer
H-ISS	Hardware-based Image-space Sheet-buffer
H-TM	Hardware-based 3D Texture Mapping

Table 1: Algorithm names and abbreviations.

non-empty voxels divided by the total number of voxels; being or not voxelized isosurfaces such as *dino* and *prostate*. Table 2 summarizes the characteristics of the different datasets. Figure 2 shows a rendered image of each of the datasets.

All the simulation results have been taken on a Pentium IV PC at 3.2 GHz with 3.5 GB of memory and an ATI Radeon X800 Pro graphics card. We have used the OpenGL library on a GNU/Linux operating system. Time is measured in seconds.

Dataset	Dimensions	Size	Type	Oc.ratio
Dino	87*39*62	210366	bool	9.04%
Prostate	194*202*256	10032128	float[4]	11.97%
Aneurism	256*256*256	16777216	char	0.37%
Skull	256*256*256	16777216	char	5.70%
Engine	256*256*128	8388608	char	20.17%
Femur	485*509*633	156265545	char	32.40%

Table 2: Characteristics of the datasets, from left to right: name, size, property value type and occupancy ratio (OR).

Table 3 shows the rendering cost of the splat projection of 16 millions of voxels using the CPU-based strategy, the 2D image-space textured *quad* and the *glPointSprite* for the different datasets (see Section 3.1). For the two latter strategies, the costs include the texture construction. As it can be observed, the hardware assisted modes are far more efficient than the CPU-based mode whose efficiency slows down as the splat size increases. The textured *quad* and *glPointSprites* have similar costs although the textured *quad* is more efficient. This difference is due to the fact that the implementation of sprites in our Linux operating system is not very efficient on Radeon cards. The same simulations on a NVidia card showed a reduction in the cost of of the *glPointSprite* to approximatively the same value as the textured *quad*.

Table 4 shows the cost of the CPU-Based buffer composition in comparison to the use of FBOs, for three different image sizes (see Section 3.2). As expected, the larger are the images, the more the FBO accelerates the buffer composition. A drawback of the FBOs is that they require the image dimensions to be powers of 2. The larger image that we have been able to process as a whole in our PC is 1024x512.

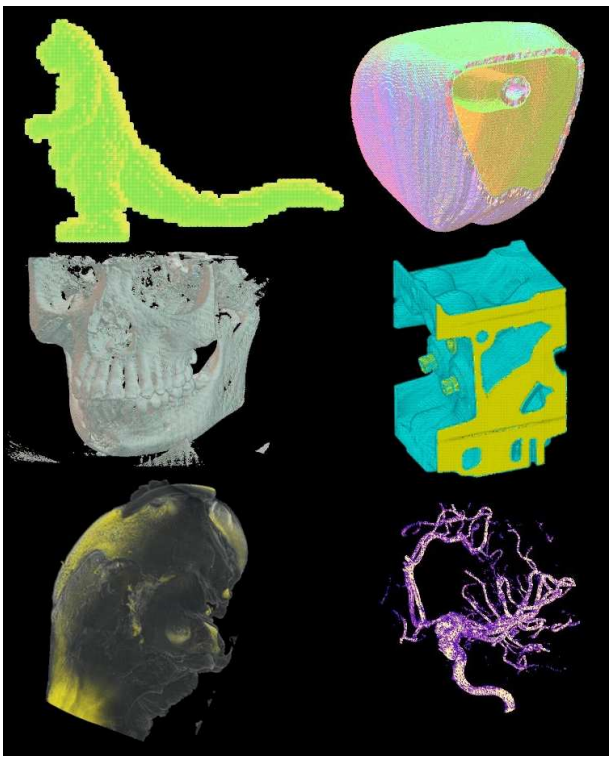

Figure 2: Rendered image of the datasets. Left column from top to bottom: dino, skull and femur. Right column from top to bottom: prostate, engine and aneurysm.

The first row of Table 5 shows the shading cost based on a LUT for volume shading or quantized pre-processed surface shading using software-based LUTs in comparison to 1D-texture LUTs. The LUT have a size of 256. Shading has been applied to 16 millions of voxels. The results are of the same order of magnitude and the CPU-based mode is even slightly better. In the second raw of the table, we show the cost of surface Gouraud shading with pre-computed gradients and one light source, implemented via CPU and with GL lighting. Again, the differences are not very significative, because the geometrical computations (one scalar product and four multiplications) are not very intensive.

Finally, Table 6 shows the cost of applying the viewing transformation to 100.000 vertices using the GPGPU program described in Section 3.4. The overall CPU-based process cost is 1.69 10^{-2} seconds. As it can be seen, the geometrical transformation cost is far more efficient in the GPU but the read-pixel operation needed to insert the voxels the buckets are a bottleneck, that globally makes the GPU assisted process slower than the CPU-based one.

Splat size	CPU-based	Quad	Sprite
3x3	19.3	5.07	6.01
20x20	411	6.06	12.44

Table 3: Costs in seconds of the splat projection: CPU-based, 2D image-space textured *quad* and *glPointSprite* for two different splat size for 16M voxels.

Image size	CPU-based	FBO
256x256	0.54	0.11
512x512	2.50	0.34
1024x512	4.90	0.64

Table 4: Cost in seconds of the buffer composition: CPU-Based and based on the FBO

The run-times in seconds of the different algorithms with different datasets are summarized in Tables 7 and 8. They all correspond to an image size of 1000x500. The values in Table 7 correspond to the plain non-optimized version of the algorithms. All the voxels be they empty or not, are processed in the rendering pipeline. Obviously, for all strategies, the more voxels, the higher the computational cost. ISS costs are comparable to OSS, but without taking into account the bucket constructions, which constitutes, indeed, a non negligible overhead. CES costs are lower than those of OSS and ISS because there is no plane composition in that method. Table 8 shows the run-time of the hardware accelerated versions of these algorithms using point-sprite, multi-texturing and FBOs. The hardware optimizations reduce between one half and two thirds the computational cost for the small and regular sized models. The *Femur* dataset rendering is not accelerated, because, being that big, the splat size is very small, thus the textured *quad* is not faster than the CPU-based access to the image buffer. These costs can be compared with hardware-assisted 3D texture-mapping. To construct the textures, we first perform a per-voxel traversal of the datasets, compute the rendered color associated to each voxel and store it in the 3D texture. Once the texture is constructed, the rendering time is very fast, at interactive framerates. This is much faster in all cases than the three hardware-assisted splatting methods, that actually perform a per-voxel classification and shading. However, the texture should be recomputed if the transfer-function or the lighting conditions change. Moreover, zoomed images from 3D-texture

Shading	CPU-based	GL
LUT	13.8	14.2
Gouraud	14.9	15.15

Table 5: Cost in seconds of the LUT-based shading in comparison to 1D-textures for a varying number of voxels.

Set-up	Transformation	Read Pixel
0.015	0.0041	0.46

Table 6: Viewing transformation using the GPGPU for 100.000 vertices: set-up, transformation and read pixels. Units are seconds

mapping exhibit a pixelized aspect whereas splatting images are smoother. The *Femur* dataset could not be rendered directly with texture mapping because of its huge size.

Dataset	CES	OSS	ISS
Dino	0.11	1.6	0.1+1.72
Prostate	3.31	7.2	5.4+7.2
Aneurism	6.99	10.1	9.5+10.2
Skull	11.21	10.8	9.3+9.7
Engine	11.30	12.5	4.5+6.5
Femur	84.30	72.4	74+59.6

Table 7: Rendering times in seconds using the different software-based splatting strategies. ISS times are split into buckets construction and splatting itself

Dataset	H-CES	H-OSS	H-ISS	H-TM
Dino	0.06	0.07	0.15+0.06	0.23+$\sim$ 0
Prostate	2.7	2.7	5.2+1.6	5.38+$\sim$ 0
Aneurism	4.8	4.8	9.3+3.7	9.45+$\sim$ 0
Skull	4.1	4.1	9.1+4.2	9.45+$\sim$ 0
Engine	3.7	3.7	4.5+2.5	4.97+$\sim$ 0.
Femur	77.0	75.5	74+54	ERR

Table 8: Rendering times in seconds using the different hardware-assisted splatting strategies and 3D texture mapping.

Tables 9 and 10 show the importance of software-based optimizations. In Table 9, we show the differences in the rendering time OSS without optimizations, with run-length encoding of non-empty voxels, hardware-accelerated and hardware accelerated with run-length. Table 10 shows the results of the same simulations for CES using a voxel array of selected voxels instead of run-length encoding as a software enhancement. The software improvements in relation to the plain non-optimized splatting depend on the occupancy ratio. The larger is the occupancy ratio, the less improvements the voxel array and the run-length provide. In general, the run-length reduces to one half the computational cost, very similarly to the hardware-based accelerations. The combination of hardware and run-length impressively reduces the cost to one tenth. The voxel array provides even better results. It is in all cases faster than the run-length. This is because the run-length has the extra cost of traversing the run-length codes.

The voxels array provides also a better speed-up than hardware accelerations, specially in the *prostate* and *aneurism* datasets, because of their low occupancy. Obviously, the maximum speed-up is obtained if the voxels array and hardware enhancements are both used. The drawback of the voxel arrays is that it requires a very large amount of memory, since it codifies the voxel coordinates, whereas this information is implicit in the run-length codification. The *femur* dataset gives the poorest results in all cases. This is because it is the larger and the one with the larger occupancy.

Dataset	OSS	OSS+RL	H-OSS	H-OSS+RL
Dino	1.6	1.54	0.07	0.03
Prostate	7.2	2.33	2.7	1.40
Aneurism	10.1	4.49	4.8	0.11
Skull	10.8	6.16	4.1	1.08
Engine	12.5	6.55	3.7	1.36
Femur	72.4	92.44	75.5	50.36

Table 9: Rendering time in seconds of OSS: basic method, with run-length encoding, hardware accelerated and hardware accelerated with run-length

Dataset	CES	CES+SEL	H-CES	H-CES+SEL
Dino	0.11	0.02	0.06	0.02
Prostate	3.31	0.7	2.7	0.65
Aneurism	6.99	0.08	4.8	0.05
Skull	11.21	1.16	4.1	0.84
Engine	11.30	1.69	3.7	1.35
Femur	84.30	61.9	77.0	48.6

Table 10: Rendering time in seconds of CES: basic method, with pre-selected voxel arrays, hardware accelerated and hardware accelerated with pre-selected voxel arrays.

6 CONCLUSIONS

In this paper, we have proposed, implemented and compared software and hardware-driven improvements of three volume splatting strategies: *composite-every-sample* (CES), *object-space sheet-buffer* (OSS) and *image-space sheet-buffer* (ISS). The analysis of the empirical results show that the three techniques have a similar processing cost, but that ISS has the overhead of the bucket computations. This is the price to pay for a better image quality and a smoother transition between successive camera positions. It should be noted, however, that ISS can be further enhanced with early splat termination. Among the hardware-based accelerations analyzed, those providing a better acceleration are the use of frame-buffer objects for image composition and textured quads

for splatting. The speed-ups are approximatively the same for CES, OSS and ISS. The creation of the buckets in ISS could also be accelerated using GPGPU if the read pixel operation is avoided. This would require storing the buckets in the GPU. We are currently investigating this possibility.

A question that is often a debate object in Visualization is that it is worth designing software accelerations for rendering, taking into account that the evolution of hardware is so fast. The results obtained in this paper show that, currently, software accelerations as those explored in this paper, provide speed-up comparable to those given by hardware-based implementations. Moreover, the combination of both types of technique can reduce dramatically the cost of rendering. Therefore, our future research focuses at designing new acceleration techniques that could be further enhanced using hardware.

Finally, software-based and hardware-based strategies still do not reduce enough the cost of rendering huge datasets. A factor that surely worsens the rendering cost of this type of datasets is how data are managed into memory. We believe that out-of-core methods specifically designed for splatting could help solving this problem.

References

[BK03] Botsh M., Kobbelt L.: High-quality point-based rendering on modern gpus. In *Pacific Graphics 2003* (2003), pp. 335–343.

[CM93] Crawfis R., Max N.: Texture splats for 3D scalar and vector field visualization. In *IEEE Visualization'93* (1993), pp. 261–266.

[CN93] Cullip T. J., Neumann U.: *Accelerating Volume Reconstruction with 3D Texture Mapping Hardware*. Tech. rep., University of North Carolina at Chapel Hill, 1993.

[Cra96] Crawfis R.: Real-time slicing of data space. In *IEEE Visualization'96* (1996), IEEE Computer Society Press, pp. 271–277.

[GK96] Gelder A. V., Kim K.: Direct volume rendering with shading via 3D textures. In *ACM Symposium on Visualization'96* (1996), Crawfis R., C.Hansen, (Eds.), pp. 23–30.

[HBH03] Hadwiger M., Berger C., Hauser H.: High-quality two-level volume rendering of segmented data sets on consumer graphics Hardware. In *IEEE Visualization '03* (2003), IEEE Computer Society Press, pp. 40–45.

[HMBG01] Hauser H., Mroz L., Bischi G., Gröller M.: Two-level volume rendering. *IEEE Trans. on Visualization and Computer Graphics 7*, 3 (2001), 242–252.

[HMSC00] Huang J., Mueller K., Shareef N., Crawfis R.: Fastsplats: optimized splatting on rectilinear grids. In *IEEE Visualization'00* (2000), IEEE Computer Society Press, pp. 219–226.

[IL95] Ihm I., Lee R.: On enhancing the speed of splatting with indexing. In *IEEE Visualization '95* (1995), IEEE Computer Society Press, pp. 69–76.

[KM01] Kilthau S., Möller T.: *Splatting optimizations.* Tech. rep., Simon Fraser University, 2001.

[Kni00] Knittel G.: The ultravis system. In *IEEE Symposium on Volume Visualization* (2000), pp. 71–79.

[KW03] Krüger J., Westerman R.: Acceleration techniques for GPU-based volume rendering. In *IEEE Visualization'03* (2003), pp. 287–292.

[LH91] Laur D., Hanrahan P.: Hierarchical splatting: A progressive refinement algorithm for volume rendering. *ACM Computer Graphics 25*, 4 (July 1991), 285–318.

[MC98] Mueller H., Crawfis R.: Eliminating popping artifacts in sheet buffer-based splatting. *IEEE Visualization'98* (1998), 239–246.

[MH01] Mroz L., Hauser H.: RTVR: a flexible *j*ava library for interactive volume rendering. In *IEEE Visualization'01* (2001), IEEE Computer Society Press, pp. 279–286.

[MHS99] Meissner M., Hoffmann U., Strasser W.: Enabling classification and shading for 3D texture mapping based volume rendering using OpenGL and extensions. *IEEE Visualization'99* (1999), 207–214.

[MSHC99] Mueller K., Shareef N., Huang J., Crawfis R.: High-quality splatting on rectilinear grids with efficient culling of occluded voxels. *IEEE Trans. on Visualization and Computer Graphics 5*, 2 (1999), 116–134.

[NM05] Neophytou N., Mueller K.: GPU accelerated image aligned splatting. In *Volume Graphics* (2005), Fujishiro I., Gröller E., (Eds.), pp. 197–205.

[OM01] Orchard J., Möller T.: Accelerated splatting using a 3D adjacency data structure. In *Graphics Interface'01* (2001), pp. 191–200.

[PHK*99] Pfister H., Hardenbergh J., Knittel J., Lauer H., Seiler L.: The volumepro real-time ray-casting system. In *ACM SIGGRAPH'99* (1999), ACM Press/Addison-Wesley Publishing Co., pp. 251–260.

[PZBG00] Pfister H., Zwicker M., Baar J., Gross M.: Surfels: surface elements as rendering primitives. In *ACM SIGGRAPH'00* (New York, NY, USA, 2000), ACM Press/Addison-Wesley Publishing Co., pp. 335–342.

[RPZ02] Ren L., Pfister H., Zwicker M.: A Hardware accelerated approach to high quality point rendering. *Computer Graphics Forum 21* (2002), 461–470.

[VHFG05] Vega F., Hastreiter P., Fahlbusch R., Greiner G.: High performance volume splatting for visualization of neurovascular data. In *IEEE Visualization'05* (2005), IEEE Computer Society Press, pp. 271–278.

[WE98] Westermann B., Ertl T.: Efficiently using graphics Hardware in volume rendering applications. *ACM SIGGRAPH'98* (1998), 169–178.

[Wes89] Westover L.: Interactive volume rendering. In *Chapel Hill Volume Visualization Workshop* (1989), pp. 9–16.

[Wes90] Westover L.: Footprint evaluation for volume rendering. *ACM Computer Graphics 24*, 4 (July 1990), 367–376.

[WG91] Wilhems J., Gelder A. V.: A coherent projection approach for direct volume rendering. *ACM Computer Graphics 25*, 4 (July 1991), 275–284.

[XC04] Xu D., Crawfis R.: Efficient splatting using modern graphics hardware. *Journal of graphics tools 8*, 4 (2004), 1–21.

[YESK95] Yagel R., Ebert D. S., Scott J. N., Kurzion Y.: Grouping volume renderers for enhanced visualization in computational fluid dynamics. *IEEE Trans. on Visualization and Computer Graphics 1*, 2 (1995), 117–132.