Image-space Sheet-Buffered Splatting on the GPU

S.Grau[1] and D. Tost[1]
[1]Divisió Informàtica Gràfica
Centre de Recerca de Enginyeria Biomèdica
UPC, Barcelona, Spain

March 15, 2007

Abstract

Image-Space Sheet-Buffered Splatting is a popular high quality volume-rendering technique specially suitable for zoomed views of the data. On the contrary to other splatting approaches, it processes the voxels in slabs perpendicular to the viewing direction. Recently, a GPU design of this method has been proposed that considerably accelerates the rendering stage. However, the bottleneck of the method is the computation of the buckets, i.e the structure handling the voxels to be rendered in each slab. This stage of the method is done on the CPU. In this paper, we propose a new design of the method that creates and manages the buckets on the GPU. The proposed method is more than twice faster than the previous ones.

1 Introduction

Splatting was originally proposed as a feed-forward algorithm for voxel-based volume datasets [Wes89]. Recently, it has gained popularity in being applied to point-based surface models [BHZK05]. The original approach considers the volume as an array of 3D overlapping kernels weighted by the voxels property values. The 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 [ZRB*04]. In the original approach of the algorithm, called *Composite-Every-Sample* (CES), the voxel footprints are composited in the image-plane one after the other in back-to-front order. Therefore, visibility is combined with reconstruction and thus, color bleeding artifacts may appear. To correct this error, Westover [Wes90] proposed the *Object-Space Sheet-Buffered Splatting*. This method adds the voxels footprints slice-by-slice into sheet planes of the voxel model most parallel to the image plane, and composites each sheet to the final image. It corrects color bleeding but it introduces noticeable popping up artifacts when the camera moves around the volume because the sheet planes change abruptly. The *Image-Space Sheet-Buffer Splatting* [MC98] solves this problem, because it uses sheet-buffers parallel to the image plane. In this approach, voxels contribute to more than one sheet, therefore different footprints corresponding to different intersections of the voxels with the sheet slab must be computed.

One of the major advantages of splatting is that only relevant voxels need to be processed. There are three main strategies that exploit this idea. First, a list of relevant voxels can be computed in a pre-process and taken as input for splatting. Examples of this approach are the *fuzzy set* [YESK95], the *ListSplat* [Cra96] and the *RenderLists* [MH01] [HMBG01] [HBH03]. Alternatively, data structures such as adjacency lists [OM01] and run-length encoding [KM01] [FPT06] can be used to skip empty and non-selected voxels. Finally, in Image-Space Sheet-Buffer Splatting, voxels can be sorted according to their value and inserted in the buckets with a fast binary search [IL95] [MSHC99].

In parallel to these software-based accelerations, several attempts have been done to speed-up splatting using hardware features. Most of the research in this area focuses on the acceleration of the kernels projection by using texture maps [CM93], 1D and 2D look-up tables *(fastSplats)* [HMSC00], Vertex Arrays and point sprites [NM05] [BK03] [VHFG05]. In addition, sheet composition can be speeded-up by using OpenGL P-Buffers [XC04] and Frame-Buffer-Objects to render the image slices. Moreover, post-shading on the slices can be implemented in a fragment shader [NM05]. Neophitou and Mueller have also proposed to increase the voxel/pixel overdraw by treating the four color channels as separate densities. Finally, they use the OpenGL depth test features combined with NVidia Depth Bounds extension in order to avoid running fragment programs on empty or opaque pixels.

Despite these improvements, the memory transfer of the data to the GPU is still the bottleneck of splatting. The optimal solution is to keep all the data into the GPU. This problem has been addressed for surface splatting [CRZP04] and particle models rendering [KLRS04] [KSW04]. However, these approaches are based on Object-Space Sheet-Buffered and Composite-Every Sample splatting. They are not directly applicable to Image-Space Sheet-Buffered Splatting, which, as mentioned above, gives the best image quality and avoids flickering artifacts. The goal of this paper is to discuss and propose a new GPU-based Image-Space Sheet-Buffered Splatting that constructs and updates the buckets on the GPU in order to avoid memory transfer between CPU and GPU. The proposed method can still benefit from the hardware-based improvements described above (point sprite, multiple density pipeline and opaque and skipping empty pixels), but in addition to them, it uses GPGPU sorting and OpenGL Depth Test to manage the buckets on the GPU.

2 Previous work

Image-Space Sheet-Buffered Splatting processes the volume by slabs parallel to the image plane [MSHC99]. Figure 1 sketches its pipeline. Each slab is associated to a data structure called *bucket* that holds the index of all the voxels that intersect the slab. Buckets are filled for each new camera position by transforming the relevant voxels with the viewing matrix. Slabs are then processed in front-to-back or back-to-front order. All the voxels of a slab are summed into a sheet buffer according to a pre-integrated kernel slice, and the sheet buffer is composed with the image buffer. No sorting is required inside the buckets since splats of a buckets are added in the corresponding sheet buffer, and adding is commutative. In order to speed up the buckets construction, a list of per-value sorted non-empty voxels can be constructed in a preprocess and used instead of the full voxel model. Other strategies to skip non-relevant voxels in the bucket construction are the use of skeletons [CSM05] and run-length encoding based on classified values [FPT06]. In principle, Image-Space Sheet-Buffered Splatting uses a post-shading scheme: it splats raw density values, computes the gradients using central differences on the projection image and applies a per-pixel shading. However, it could also support pre-shading. Post-shading provides images with less blur on the objects edges [MMC99] but it is only applicable for transfer-function based classification, and it is not suitable for segmented tagged data.

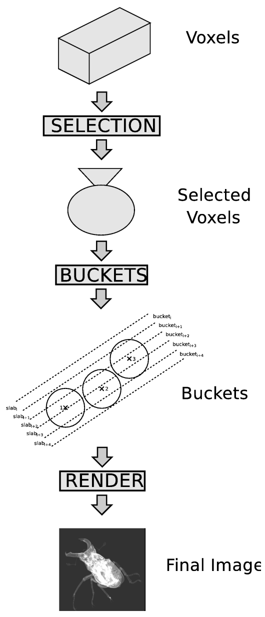

Figure 1: Image-Space Sheet-Buffered Splatting Pipeline.

In a recent paper, Neophitou and Mueller [NM05] have proposed a GPU based acceleration of Image-Space Sheet-Buffered Splatting. They keep the buckets construction step on the CPU and use the GPU in the second stage of the pipeline. Their approach brings several contributions. First, instead of rasterizing a textured polygon per splat, they use the *Point Sprites* extension that requires only one vertex per splat to be sent. They also use the Vertex Arrays extension to pack the points of the slices. Next, they use a post-shading process, which allows us them to splat 4 slices at a time by using each of the four color channel RGBA for a different density slice. In addition, they use the early z-rejection test with two purposes: to avoid splatting on pixels of the slice that are already opaque (early splat elimination), and to restrict shading and compositing only to the pixels of a slice that have been touched during splatting. Putting all together these accelerations, they report between 2fps (frame per seconds) for the semi-transparent $1,2M$ effective splats engine data set and 10fps for the $219K$ effective splats Stony Brook lobster on a Pentium IV with NVidia Quadro FX 3400.

The current limitation of this implementation is, as reported by the authors themselves, the memory transfer between the CPU buckets and GPU slices. Keeping the data inside the graphics board have been investigated Composite-Every-Sample and Object-Space Sheet buffered Splatting but not for Image-Space Sheet-Buffered Splatting. Specifically, Chen et al. [CRZP04] have proposed a GPU acceleration of EWA splatting [ZPvBG01] for the object-space sheet-buffered approach. They store the volume data on the GPU sorted according to the object-space slices along with a 4-vertices quad proxy geometry per voxel. In order to reduce the memory requirements, they compress the proxy geometry and decompress it on the fly in the vertex shader. Vega-Higuera et al. [VHFG05] propose to classify the data in a pre-process and send to the GPU the center point of only the selected voxels. This approach requires thus to depth sort the points for transparent rendering. To speed this part of the pipeline, sorting in done on the CPU, after classification in the pre-process according to 8 viewing directions which correspond to the octants of the 3D space. The indices of the sorted voxels are stored in 8 Vertex Buffer Objects and used in the splatting stage. This strategy provides high performance, specially on low occupancy ratio neurovascular data, which were the motivation of the paper.

GPU-accelerated splatting of points and particle systems shares the problem of memory transfer between CPU and GPU. In the context of computer games simulation, Latta [Lat04] has proposed to store the positions and velocities of the particles in floating point textures. In order to update them, he uses a pair of texture for both parameters and a double buffering technique to switch between textures. In order to perform alpha-blending, particles are sorted on the GPU with the *Odd-Even Merge Sort* distributed over 20 to 50 frames. If enough frame-to-frame coherence exists, which is probable in computer games and simulations, this progressively sorting approach gives visually acceptable images. The same sorting technique is used for large particle systems [KLRS04]. Kipfer, Segal and Westermann [KSW04] designed the Uberflow particle system that exploits the *Super Buffers* Open GL extension. They propose an improvement of the GPU implementation of the Bitonic Merge Sort designed by Buck and Purcell [BP04]. The technique is explained with more details in another paper [KW05]. Krüger et al. [KKKW05] re-use this technique for the visualization of 3D flows. They store the position of the particles in the RGB channels of floating point texture and a parameter representing the lifetime of the particle in the alpha-channel. Particles are rendered using transparent point sprites. Sorting is performed with the GPU Bitonic Merge sort.

The goal of this paper is to propose an Image-Space Sheet-Buffered Splatting that makes full use of the GPU computational resources and computes the buckets on the GPU. Our work is inspired on Neophitou and Mueller's work [NM05] and on the rendering particle systems approaches described above.

Next, we overview the proposed strategy 3.1, and then we describe with more details each step of our method.

3 Image-space Sheet-Buffered Splatting on the GPU

3.1 Overview

Figure 2 illustrates the proposed pipeline. We start loading the selected voxels on a 2D texture *(Voxels Texture)* and next, we transform all the voxels according to the viewing system. The result is stored in the *View Transformed Voxels Texture*. The problem that arises then is how to compute the buckets. Scattering, i.e. random writes to specified addresses, is not efficient on GPUs. Fragment shaders cannot change the writing texture position. Thus, it is not feasible to process sequentially the voxels and insert them in texture buckets according to their z-value, as it is done in the CPU pipeline illustrated in Figure 1. What we do instead is depth sorting the transformed voxels. The resulting texture *(Depth-Sorted Voxels Texture)* is an implicit representation of the buckets since, as indicated in Figure 3, slabs correspond to overlapping sorted subsequences of voxels in the texture. Therefore, in order to process sequentially the slabs, we need to know the size of each of these subsequences. We define the bucket of a slab as the subset of the depth sorted voxels of the texture whose center falls inside the slab (see Figure 4). This is different from the CPU approach, because these buckets do not contain all the voxels intersecting the slab, but only a subset of them. Therefore, each voxel belongs to only one of these buckets, although it is splatted for all the slabs it intersects.

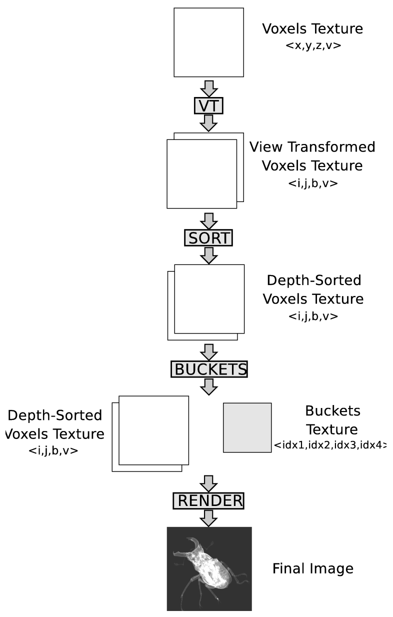

Figure 2: GPU-based Image-Space Sheet-Buffered Splatting Pipeline.

The next stage after depth sorting the texture is the bucket size computation. It gives as a result a texture containing the position of the last voxel of each bucket *(Buckets Texture)*. The last step of the pipeline consists of processing the slabs one after the

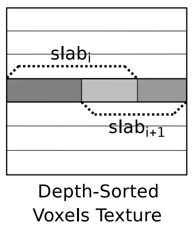

Figure 3: Voxels of a slab in the *Depth-Sorted Voxels Texture*. Consecutive slabs correspond to overlapping subsequences of the texture.

other. The radius of the voxels kernel determines the number of previous and following buckets that affect a slab in addition to its own bucket.

If the camera is static and the transfer function changes, only the render stage needs to be re-done. When the camera moves, the pipeline is executed starting by computing the *View Transformed Voxels Texture*. In these two cases, the *Voxels Texture* doesn't change and it is kept in the GPU. If the selection pre-processing is modified, the *Voxels Texture* should be recomputed. Alternatively, we can load all the non-empty voxels on the GPU in order to avoid the CPU-GPU transfer. In this case, the selection is done using an opacity transfer function. If the number of non-empty voxels is large, the texture memory may be insufficient to load them all. In order to solve this problem, we have developed a bricking strategy based on the use of multiple textures.

3.2 Data structures

As shown in Figure 2, the basic data structures of our method are the 2D textures that store the voxels throughout the different steps of the pipeline. Each voxel is encoded in a texel position, storing its coordinates in the RGB channels and its value in the alpha channel ($< x,y,z,v >$). In order to store the voxel coordinates with the maximum precision, we use the FP32RGBA format that provides 32 bits floating point for each channel.

Textures are used together with the Frame Buffer Objects extension (FBO) in order to apply the ping-pong technique. This technique is used in recursive processes that need to re-use the last computed values in the next stage of recursion. It uses two textures attached to an FBO by color attachment. When a stage begins, the texture used as source in the previous stage is set as the target, and the previous target texture is the current source texture. Specifically, we use the ping-pong technique in the sorting stage to compute the *Depth-Sorted Voxels Texture*.

In addition to FBO, we use Vertex Buffer Objects (VBO) with the Render to Vertex Buffer (R2VB) technique that lets us create VBO using the values of a 2D texture. We use the Pixel Buffer Object (PBO) extension to load the 2D texture values onto the VBO. This keeps all the data flow inside the GPU. We use R2VB in the *Buckets Texture* construction and for the rendering stage.

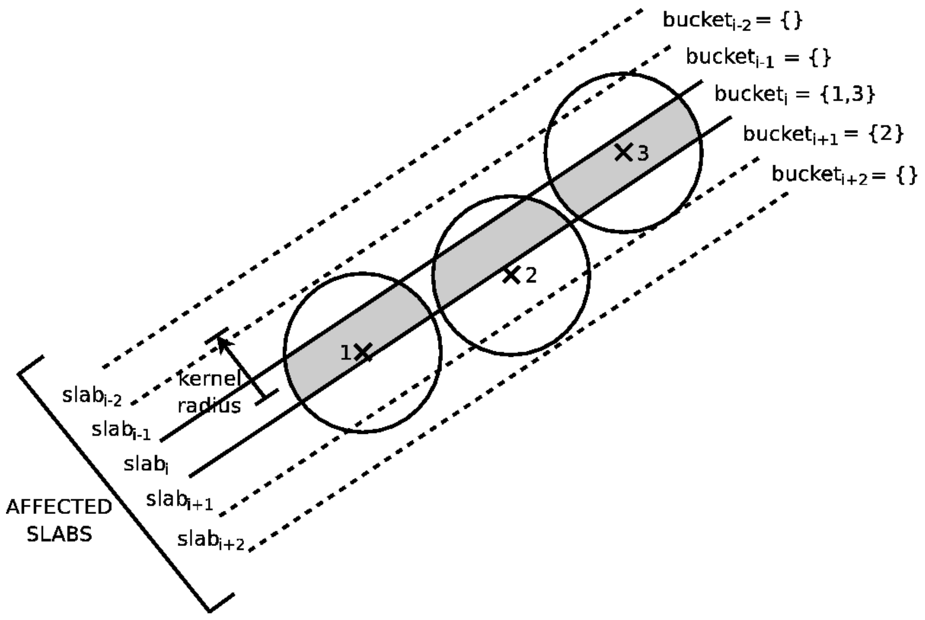

Figure 4: Each bucket of a slab has the subset of the depth sorted voxels whose center falls inside the slab. *Bucket$_i$* has only voxels 1 and 3. For the rendering stage voxels of *bucket$_i$* will be taken into account in $slab_{i-2}$ to $slab_{i+2}$.

3.3 Viewing Transform

This stage computes the *View Transformed Voxels Texture* from the *Voxels Texture*. We use a fragment shader that transforms the coordinates of the voxels stored as RGB values in the source texels according to the viewing matrix. For each texel, the fragment shader stores the 2D raster coordinates of the voxel's projection in the R and G channel of the target texture, its depth value in the viewing coordinate system in the G channel and the voxel property value in the alpha channel ($< i, j, b, v >$).

3.4 GPGPU sorting

Once the points have been transformed into the viewing coordinate system, they need to be sorted according to the distance to the viewer in order to define the volume slabs.

Sorting on the GPU uses the programmable graphics hardware as a stream processor. Therefore, the challenge of designing GPGPU sorting algorithms is to exploit as best as possible the parallel nature of the GPU architecture. Data-driven sorting algorithms such as Quicksort, that are the fastest on CPU, are not suitable for GPU implementation. This is why existing GPU sorting methods, basically *Odd-Even Merge Sort* and *Bitonic Sort*, are network sorting data-independent strategies. Current GPU implementations of *Bitonic Sort* are much faster than *Odd-Even Merge Sort*. However, this latter strategy is sometimes preferred because it gives a partial ordering at each step.

This may be sufficient in applications such as particle systems which spread sorting throughout various frames. In our case, we use *Bitonic Sort*, because we need an exact ordering at each frame and thus, the sort-while-drawing property is not convenient.

Govindaraju et al. [GRHM05] have improved the first implementations of *Bitonic Sort* [**?**] [KW05] by using cache-efficient memory accesses. This sort achieves very high performances: on an NVidia GeForce 7800, it is about twice as fast as the best quick sort CPU implementation. In addition, it has been adapted to sort very large billion record databases [GGKM06]. Very recently, Gress and Zachmann [GZ06] have designed an Adaptive Bitonic Sort *(ABIsort)* that has an optimal complexity of $O((nlogn)/p)$, being p the number of stream processor units and n the number of values to be sorted. These authors report that *ABisort* is the fastest GPU method. However, in our implementation we have preferred the Govindaraju et al.'s method, because their library is fully accessible and easy to adapt to our application.

The library sorts an array of $< key, value >$ pairs. It uses the *Multiple Render Target* (MRT) extension to efficiently store the keys and the values. One render target stores the keys and the other the values, and a one-to-one correspondence exists between the two textures. This allows us keeping four instances in one RGBA texel. The sorting steps use the ping-pong strategy described above.

The library takes as input a CPU array. In our case, the view transformed voxels to sort are already in the GPU. The sorting key is the depth voxel value b. We use a fragment shader on the *View Transformed Voxels Texture* that fills the render targets with the b texel values as the key and a voxel index in the texture as the value. Once sorting is finished, another fragment shader fills the *Depth-Sorted Voxels Texture* from the sorted render targets.

3.5 Bucket size computation

In this stage of the pipeline, we compute on the GPU the *Buckets Texture* that stores the position of the last voxel of each bucket *(bucket boundary voxel)*. This texture cannot be computed directly from the *Depth-Sorted Voxels Texture* using fragment shaders because there is no relationship between these two textures. The former contains all the selected voxels, whereas the latter contains only one voxel per bucket. Therefore, we need a vertex shader capable of writing at specified positions on a texture. However, determining if a voxel is bucket boundary requires to compute its bucket and compare it to the bucket of the following voxel in the *Depth-Sorted Voxels Texture* (see Figure 5). This would be very expensive in a vertex shader, because it requires a texture fetch for each voxel, and current vertex shaders do no implement efficiently this operation. For this, we have splitted the computation of the *Buckets Texture* into two steps: first, we compute the bucket boundary voxels using a fragment shader and next, we construct the *Bucket Texture* using a vertex shader.

In the first step, we render the *Depth-Sorted Voxels Texture* into an auxiliary texture that indicates for each voxel if it is bucket boundary or not. This auxiliary texture stores for each voxel the coordinates of its corresponding bucket in the *Bucket Texture* in the R and G channels. In the B channel, it stores a false z value inside the viewing frustum if the voxel is bucket boundary and outside otherwise. The alpha channel stores the index of the voxel in the *Depth-Sorted Voxels Texture* (see Figure 6).

The second step uses this auxiliary texture as a VBO and renders it using glPoints of size one. The points corresponding to non bucket boundary voxels are rejected in the early-depth test. Thus, only the bucket boundary voxels are rendered in the *Bucket Texture*. The shader writes in the texture only the index of the voxel in the *Depth-Sorted Voxels Texture*.

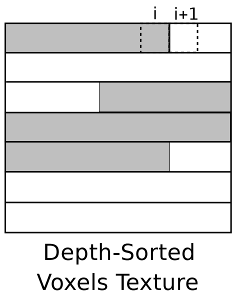

Figure 5: Bucket boundary voxels. Voxel *i* is at the boundary of the first bucket, because the center of the next voxel $(i+1)$ falls inside the next slab.

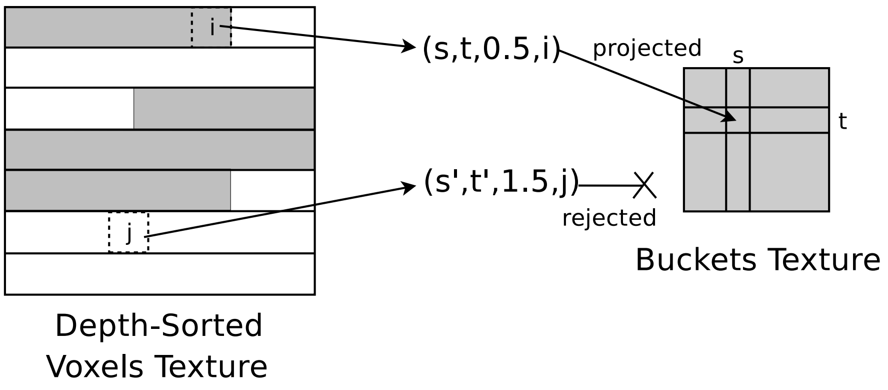

Figure 6: Buckets Texture construction. The bucket boundary voxel *i* has a 0.5 z value in the auxiliary texture, and thus, it is rendered in the *Buckets Texture*. The non-bucket boundary voxel *j* is rejected.

3.6 Rendering

The rendering process needs the size of the buckets and so, the *Bucket Texture* is transferred back to the CPU. This is a very low cost operation, because the texture size is small (the number of buckets or slabs). Then, we have the distribution of the buckets on the CPU and, on the GPU, the depth sorted voxels. Again, we use the R2VB extension to create a VBO that contains all the voxels. In order to render $slab_i$, we splat all the voxels of the buckets that may intersect it: from bucket $bucket_{i-radius}$ to $bucket_{i+radius}$, being *radius* the radius of the voxel's kernel. This is illustrated in Figure 4. To render all the voxels of a bucket, we use *DrawArray* with the first voxel's position and the number of voxels of the bucket.

Starting at this point, the rendering pipeline proceeds as in previous GPU implementations of Image-space Sheet-Buffered Splatting. The voxels are splatted using the *glPoint* primitive with the Point Sprites extension. This extension allows us to create an automatic quad from a *glPoint*, and thus, it reduces to one instead of four the vertex traffic. Voxels have a different kernel footprint in the different slabs that they intersect. Neophytou and Mueller [NM05] propose to store only one footprint and to modulate it with an appropriate slab coefficient. Alternatively, we propose to compute all the different kernel footprints and to store them all in one 2D texture. When a voxel is splatted, an index to its footprint is computed and used to determine the coordinates of the sub-texture containing the corresponding footprint. This is trivial to do for splatting with texture quads, but a little more difficult using Point Sprites, since with this extension the texture coordinates are computed automatically. For Point Sprites, the correct texture coordinates must be computed in a vertex shader.

4 Brick Processing

The limitations of our method can come from the texture size which is limited and may not always fit an entire volume. In our case, the memory limitation is 4096x4096. This allows us to render up to 16M selected voxels, which is a reasonable model size.

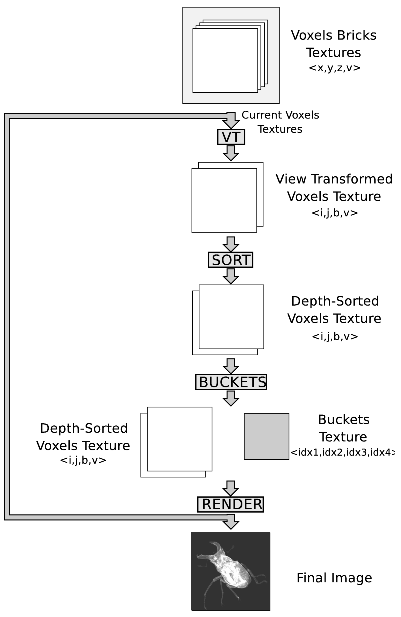

Figure 7: The proposed GPU Image-Space Sheet-Buffered Splatting Pipeline.

However, when larger models need to be processed, we subdivide them using bricks [**?**]. Each brick is stored on the GPU as a 2D texture *(Voxel Bricks Texture)*. The maximum number of voxels of each brick is given by the texture memory size. The new pipeline is represented in Figure 7. Bricks are computed by subdividing the volume into octants. They are traversed orderly according to the camera position.

A drawback of bricking is that overlapping kernels at the boundary between bricks can be composed in incorrect order. However, bricking is needed when the number of selected voxels is large, and thus the ratio pixels per voxel is generally low. In this case, kernels do not overlap very much and artifacts at the brick boundary are not perceivable.

For very large datasets such that all the bricks cannot be loaded in GPU-memory, before processing a brick, we check if it is already in the GPU. Otherwise, we need to transfer it from the CPU.

5 Results

Table 1 shows the results of the proposed method for different datasets. Color plates of the rendered images are included in Figure 8. In order to evaluate the improvement of our strategy, we have compared it with an implementation that computes the buckets on the CPU and uses the GPU only in the second part of the pipeline. The simulations have been performed on a Pentium Dual Core 3.2 GHz with 3.2 Gb memory and a NVidia GeForce 7900 GTX with 512 Mb memory. The results show that our method accelerates rendering in a factor between 2 and 2.5.

In all the simulations, we have seen that the most expensive stage of the pipeline is sorting. Therefore, even better performance can be expected when new GPU-based sorting methods appear.

Data set	Size	Effective Voxels	CPU FPS	GPU FPS
Lobster	324x301x56	233K	13.2	35.4
Engine	256^2x128	1.3M	2.4	5.6
BostonTeapot	256^2x178	4.9M	0.8	1.8
Aneurism	256^3	169K	18.4	43.4
Bonsai	256^3	1.3M	2.4	5.6
StagBeetle	832^2x494	13.8M	0.4	0.8

Table 1: Results. The efficiency of rendering is measured in frames per second. It is computed for a visualization sequence with a moving camera. All the images are rendered with semi-transparency. Frames per second in the CPU column correspond to an implementation of the method that uses GPU only for the rendering part. GPU results correspond to our pipeline.

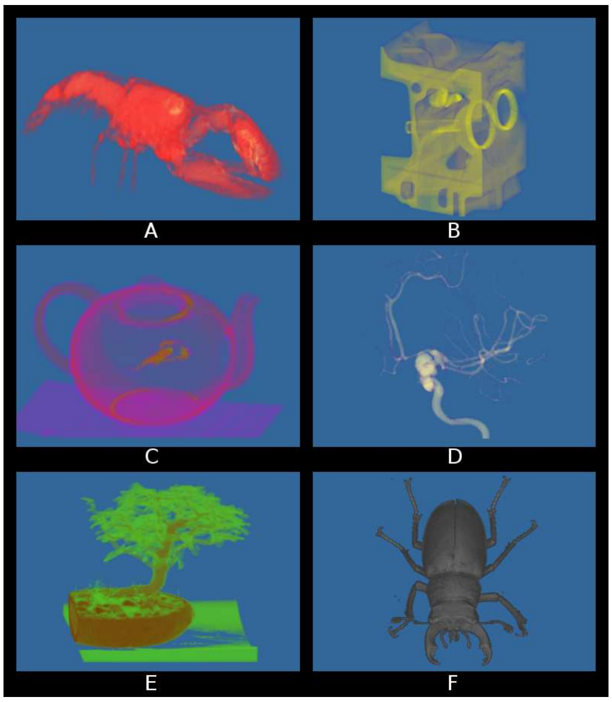

Figure 8: Example renderings using the Image-space Sheet-Buffered Splatting on the GPU. (A) Lobster, (B) Engine, (C) Boston Teapot, (D) Aneurism, (E) Bonsai, (F) StagBeetle.

6 Conclusions

In this paper, we have proposed a new Image-space Sheet-Buffered Splatting design that creates and manages the buckets on the GPU. Our implementation achieves frame rates twice and more faster than methods that use the GPU only for the rendering part. Moreover, the evolution of the speed of GPU algorithms is dramatical in relation to the evolution of CPU algorithms. Therefore, as GPUs will evolve, the cost of the most expensive part of our pipeline, sorting, will reduce, and thus, much higher rendering speed-ups are expected.

Our future research is to try to adapt this method to multi-modal data that require performing a fusion of the property values of the different modalities either in the splatting stage or during the buckets construction depending on if post-shading or pre-shading is applied. Furthermore, we will investigate how to render time-varying data using this strategy taking into account frame-to-frame coherence.

References

[BHZK05] BOTSCH M., HORNUNG A., ZWICKER M., KOBBELT L.: High quality surface splatting on todays GPU. In *EG Symp.on Point-based graphics* (2005), Pauly M., Zwicker M., (Eds.), pp. 1–8.

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

[BP04] BUCK I., PURCELL T.: *A Toolkit for Computation on GPUs*. Addison-Wesley, 2004, ch. GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics, R. Fernando Editor.

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

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

[CRZP04] CHEN W., REN L., ZWICKER M., PFISTER H.: Hardware-accelerated adaptive EWA volume splatting. In *IEEE Visualization'04* (2004), pp. 67–74.

[CSM05] CORNEA N. D., SILVER D., MIN P.: Curve-skeleton applications. In *IEEE Visualization* (2005), p. 13.

[FPT06] FERRÉ M., PUIG A., TOST D.: Decision trees for accelerating unimodal, hybrid and multimodal rendering models. *The Visual Computer*, 3 (2006), 158–167.

[GGKM06] GOVINDARAJU N., GRAY J., KUMAR R., MANOCHA D.: Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD Conference* (2006), pp. 325–336.

[GRHM05] GOVINDARAJU N. K., RAGHUVANSHI N., HENSON M., MANOCHA D.: *A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors*. Tech. rep., UNC Tech. Report, 2005.

[GZ06] GRESS A., ZACHMANN G.: GPU-ABiSort: Optimal parallel sorting on stream architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (April 2006).

[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.

[KKKW05] KRUGER J., KIPFER P., KONDRATIEVA P., WESTERMANN R.: A particle system for interactive visualization of 3d flows. *IEEE Transactions on Visualization and Computer Graphics 11*, 6 (2005), 744–756.

[KLRS04] KOLB A., LATTA L., REZK-SALAMA C.: Hardware-based simulation and collision detection for large particle systems. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2004), ACM Press, pp. 123–131.

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

[KSW04] KIPFER P., SEGAL M., WESTERMANN R.: Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2004), ACM Press, pp. 115–122.

[KW05] KIPFER P., WESTERMANN R.: *GPU Gems 2, M. Pharr Editor*. Pearson Education, 2005, ch. Improved GPU sorting.

[Lat04] LATTA L.: *Game developers Conference*. Gamasutra, 2004, ch. Building a million particle system.

[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.

[MMC99] MUELLER K., MÖLLER T., CRAWFIS R.: Splatting without the blur. In *IEEE Visualization'99* (1999), pp. 363–371.

[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.

[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.

[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.

[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.

[ZPvBG01] ZWICKER M., PFISTER H., VAN BAAR J., GROSS M.: EWA volume splatting. In *IEEE Visualization'01* (2001), pp. 29–36.

[ZRB*04] ZWICKER M., RASANEN J., BOTSCH M., DACHSBACHER C., PAULY M.: Perspective accurate splatting. In *Graphics interface'04* (2004), pp. 247–254.