Stochastic Particle-Based Volume Rendering

Philip Voglreiter*
Supervised by: Bernhard Kainz†

Institute for Computer Graphics and Vision
Graz University of Technology
Graz / Austria

Abstract

In this paper we propose a particle-based volume rendering approach for unstructured three-dimensional tetrahedral polygon meshes. We stochastically generate millions of particles per second that are projected on the screen in real-time. In contrast to previous rendering techniques of tetrahedral volume meshes, our method does not need a prior depth sorting of geometry. Instead, the rendered image is generated by choosing particles closest to the camera. Furthermore, we use spatial superimposing. Each pixel is constructed from multiple subpixels. This approach not only increases projection accuracy, but allows also a combination of subpixels into one superpixel that creates the well-known translucency effect of volume rendering. We show that our method is fast enough for the visualization of unstructured three-dimensional grids with hard real-time constraints and that it scales well for a high number of particles.

Keywords: volume rendering, GPGPU, particle-based, object space

1 Introduction

Volume rendering is used in many disciplines. Visualization of medical data or simulated data arising from finite element methods are just a few examples. The data to be rendered can be represented as regular or irregular structure. Regularly structured data originates mostly from medical imaging devices (MRI, CT, etc.). Direct visualization of these volumetric datasets is well researched and various methods exist. A good overview over standard methods is given by Hadwiger *et al.* [8].

Irregular datasets – or unstructured grids –, are mainly used for simulations, for example for finite element analysis [3], which normally uses an input of irregular shape and which requires connectivity information of the grid's nodes. Rendering such grids is an ongoing field of research. Early approaches use standard geometric algorithms such as plane sweep techniques. Other algorithms directly exploit the grid structure. Using tetrahedral grids is the most prominent method. The grids can either be projected directly, or they can be rendered in a preprocessed state to speed up the rendering process [14].

Basically, all methods for volume rendering can be divided into two main areas. They either are *image-based* or *object-based*. Image-based methods, like ray casting, generally scale with image resolution. Their performance highly depends on the amount of pixels to be displayed. In contrast, object-based approaches like point splatting are less dependent on image resolution. The complexity of rendering is strongly tied to volume complexity. Particle-based volume rendering (PBVR) belongs to the object space approaches. In contrast to many other methods in this category, PBVR does not require depth sorting of any kind. Instead, we treat projected particles in a way that is similar to z-buffering.

Modern applications demand fast visualization techniques. Real-time generation of images with an acceptable frame rate is essential for visualization techniques such as Augmented Reality [5] or other applications with hard real-time contstraints. Often, several tasks need to be performed in parallel. Especially medical applications need to provide a wide field of techniques concurrently to the visualization of data. Recorded images often need to be segmented. Also, simulations need to be performed simultaneously to visualization.

Modern GPUs give an option for fast visualization methods and allow solving a vast amount of parallel problems in real-time. In this paper, we introduce a novel way of stochastic PBVR on modern GPUs. In contrast to the highly sophisticated particle generation methods (Metropolis [12]) used by former approaches, we introduce a method of particle generation with little computational effort. Our proposed method also allows online control of the number of generated particles. This is crucial for applications with hard real-time constraints and allows to alter visual effects such as density during runtime. Because the number of particles also influences the computational effort and memory consumption, our proposed online control can also be used to steer the use of resources. This is necessary for applications that require an execution of different critical GPU accelerated tasks concurrently.

Furthermore, most PBVR methods for unstructured grids do not take final opacity values of rendered volumet-

*philip.voglreiter@student.tugraz.at
†kainz@icg.tugraz.at

ric cells into account. On the one hand, many approaches do not allow this because of inherent problems of the particle generation method. On the other hand, most methods consider particles as completely opaque primitives. Thus, the opacity of a particle is neglected.

Contribution We provide a fast, on the fly method for parallel, real-time particle generation in tetrahedral grids and simultaneous rendering. The proposed method prevents visual patterns such as streaks or clusters in the final images. We also introduce an improved method for particle superimposing. Thereby, we address perceptive issues occurring at different depth levels of the rendered volumes.

2 Previous work

In [2], Avila *et al.* propose an approach of direct volume rendering and define a rendering pipeline for irregular datasets. They refer to the plane-sweep technique, which is widely used to solve geometrical problems. Shirley *et al.* [17] first describe a method of projecting tetrahedrons onto the image plane. The tetrahedrons need to be sorted before projection. Sorting is known to be $O(n\ log\ n)$ for most sorting algorithms, and thus a larger grid size means more computational effort.

Approaches like projected tetras have already been implemented on the GPU [11]. The authors exploit the capabilities of shaders and CUDA to perform depth sorting of the tetrahedrons. Sorting large numbers of tetrahedrons is a time-consuming task and can be inefficient. As alternative approach, Challinger [6] describes a method for ray casting of unstructured grids. Ray casting generates images of a higher quality but shows an $O(n^3)$ complexity in the worst case. However, ray casting offers ways to benefit from modern GPU capabilities as was shown in [21]. Still, the rendering itself requires a high computational effort and is usually too slow for real-time applications.

Point splatting [20] is a method very similar to particle-based approaches. The authors show an efficient way to generate oval splats with low memory cost, but point splatting inherently produces artifacts in the rendering process.

In [16], Sakamoto *et al.* describe a general approach of PBVR. They base their particle generation algorithm on the Metropolis Method [12]. The Metropolis method is a well-known, efficent Monte-Carlo algorithm [13] for random number generation. Generally, the Metropolis method is rather inefficient concerning computational speed. In [15], the authors go deeper into detail of their particle generation method. Also, they consider rendering tetrahedral grids by voxelizing them. Voxelizing a tetrahedral grid can be rather time-consuming, depending on the vertex distribution. Vertices, which are not located exactly at corner points of the rectilinear grid, which characterizes a voxelized volume, need to be interpolated. But the voxel values need to be interpolated again for actual rendering. Interpolation inherently produces erroneous results. Interpolating interpolated values increases the amount of error even further. Pelt *et al.* [19] use a particle-based method to perform illustrative volume rendering. They describe hatching and stippling techniques using particles and also visualize contours of datasets with their method.

3 Particle-based volume rendering

The main idea of PBVR is to construct a dense field of light-emitting, opague particles inside a volumetric dataset. These particles are used to perform object-based rendering by simulating the light emission of particles. Mutual occlusion induced by completely opague particles plays a major role during rendering. Sakamoto *et. al.* [16] describe the basic model in more detail. Generally, PBVR involves two major steps. First, a proper particle distribution inside the volume needs to be generated, which is described in Section 3.1. Second, in Section 3.2 we outline how the particles are projected onto the image plane. In these two sections, we give a detailed description of those two steps as well as some detail on methods to increase the visual performance of the algorithm.

3.1 Particle Generation

In this paper we use a stochastic process to generate the field of particles. It is desirable to generate particles uniformly distributed over the whole volume. This results in images without visual artifacts, namely streaks, holes, or clusters. We split the volume into tetrahedral cells and perform particle generation per cell. This divide and conquer approach has several effects. On the one hand, particle generation is parallelizable. On the other hand, the generated particles do not necessarily resemble a uniform random distribution over the whole volume anymore. Thus, we will show how to treat this situation effectively in the following paragraphs.

Particle Distribution over Cells We consider a maximum number of particles p_{max} for the whole model. This number comprises the maximum amount of particles to be rendered throughout the whole volume. Note that the maximum amount of particles is rarely fully exploited. To accomplish a visually acceptable distribution of particles, we need to determine the amount of particles p_{cell} that each cell may project. We calculate this number by using the proportion of cell volume V_{cell} to the total volume of the grid V_{grid}. This ratio directly describes how many particles of the total quantity we may use for a given cell. Therefore, the number of particles per cell is

$$p_{cell} = V_{cell}/V_{grid} * p_{max}. \quad (1)$$

This can be proven easily for one dimension. Generalization to the third dimension thereafter is trivial, but unfortunately both exceed the length of this paper.

Using this method, we generate one single dense distribution for each cell. What we actually want to achieve is a uniform distribution over the whole volume. Our method of splitting the whole volume into separate sub-volumes shows a statistical benefit. In short, we can distribute the particles over the cells in a way that resembles the distribution of the mean values of the generated particles for different spatial regions. The mean value of the generated particles in a cell – considering a distribution over the whole volume rather than over single cells – is exactly the proportion of the cell volume V_{cell} to the total volume V_{grid}. Thus, we may generate the particles per cell and still statistically achieve a uniform distribution over the whole data space.

Particle Position For the positions of the particles, we use barycentric coordinates on the tetrahedral cells. Barycentric coordinates describe a point that is guaranteed to be within the borders of a given polygon. In case of tetrahedrons, the barycentric notation of a point inside it is

$$P = \alpha * V1 + \beta * V2 + \gamma * V3 + \delta * V4 \tag{2}$$

where V1,V2,V3 and V4 denote the corner points of the tetrahedron and α through δ resemble the barycentric parameters. However, some constraints apply to these parameters. First, each parameter must be greater than zero. Second, all four parameters must sum up to one. Thus, we can rewrite the parameter δ to

$$\delta = 1 - (\alpha + \beta + \gamma) \tag{3}$$

and after replacing δ in Equation 2, the barycentric description of a point results in

$$P = \alpha * V1 + \beta * V2 + \gamma * V3 + (1 - (\alpha + \beta + \gamma)) * V4 \tag{4}$$

This means that we only need to randomly generate parameters α , β and γ for each particle. By using Equation 3, we can calculate δ directly.

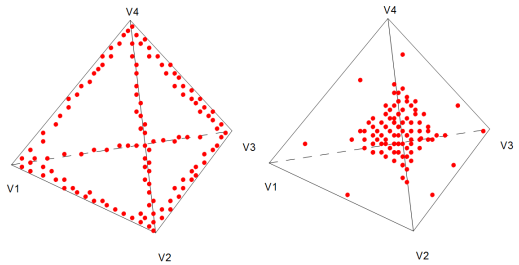


Figure 1: Generated particles clustered near the edges of a tetrahedron on the left and near the center on the right. This effect is created due to using an incorrect distribution

There are several ways to generate barycentric coordinates randomly. Many of those approaches possess statistically correct mean values but still introduce disturbing visual patterns. The most straight-forward way is random generation of all four parameters and dividing them by their sum. This leads to a statistically recorded mean value of 0.25 for each parameter. But the parameters are not statistically independent if generated this way. The following *Definition* shows the computation of the parameters:

Definition 3.1 *Barycentric parameters generated as random numbers over the interval* $(0,1)$ *and their expected values after applying the barycentric summation constraint are given as*

$$E(x) = \frac{0.5}{E(\alpha) + E(\beta) + E(\gamma) + E(\delta)} \tag{5}$$

$$x \in \{ \alpha, \beta, \gamma, \delta \} \tag{6}$$

However, the particles tend to concentrate in the cell centers, which leads to disturbing visual effects. The border regions of the cell remain very sparse. Another method involves parameter generation within the mentioned constraints. After each parameter is generated, the remaining maximum is updated and used for the generation of the next parameter. This leads to the mean values

$$E(\alpha) = 0.5, E(\beta) = 0.25, E(\gamma) = 0.125, E(\delta) = 0.125 \tag{7}$$

To equalize the distribution, parameters can be shuffled randomly. This circumvents the situation of the first parameter averagely using half of the parameter range and leads to statistically recorded mean values of 0.25 for each parameter. But this method suffers a problem similar to the one of the straight-forward method. The generated stochastic variables are not independent. This approach produces the opposite of the simplistic generation. Cell centers are sparsely covered with particles and cell borders show a strong visual pattern. Both problematic methods of particle generation are illustrated in Figure 1

A method to generate particles with a statistically correct, patternless distribution was introduced by Glassner [7]. The method bases on folding geometry. The barycentric parameters are randomly generated in a parallelepiped, which comprises of the desired tetrahedron and its mirrored counterparts. After generation, the parameters are fitted to the desired tetrahedron. In detail, we first randomly generate the parameters α , β and γ within the range $(0,1)$. In the next step we calculate $sum = \alpha + \beta + \gamma$. If that sum is greater than one, we need to manipulate the generated parameters as we violate the barycentric constraint of parameter summation equaling one. Therefore, we compute $p = 1 - p$ for each parameter. In the final step we calculate $\delta = 1 - (\alpha + \beta + \gamma)$. Figure 2 shows a proper particle distribution achieved by the described method.

Summarizing, the particles are generated uniformly distributed in a parallelepiped. Points which are outside the

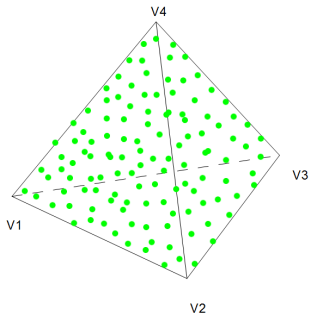


Figure 2: Uniform random distribution of particles over a tetrahedral cell showing no visual patterns

tetrahedron are transformed inside. This method generates a patternless uniform random distribution of coordinates within the constraints of barycentric parameters.

Particle Scalar Value So far we are able to generate a cloud of uniformly random distributed particles. The scalar value corresponding to each particle is easy to calculate. We already know the geometric influence of each corner point to a given particle, namely the barycentric coordinates. We can reuse those parameters to calculate the scalar value of a particle as

$$s = \alpha * s1 + \beta * s2 + \gamma * s3 + (1 - (\alpha + \beta + \gamma)) * s4, \quad (8)$$

where s1 through s4 denote the scalar values of the corner points.

Particle Emission Probability In section 3.1 we describe a uniform particle distribution over the whole grid. Simply projecting all generated particles would lead to a high density of particles hitting the screen regardless of cell opacity. So we need to thin out the particle field. As we still want to avoid patterns within the rendered images, we do this stochastically.

Using the scalar value and a transfer function, we first determine the opacity that a particle would anticipate. Based on this calculated opacity, we use the rejection method [13] to decide whether a particle is emitted or not. Generally speaking, the determined opacity of a particle op, which is in the interval $(0,1)$ describes the emission probability. Corresponding to each particle we next generate a stochastic variable x within the interval $(0,1)$ on the real line. Now we perform an emission check, i.e. if x is smaller than op, the particle is accepted and emitted. Otherwise, the particle is discarded. When applying this method, cells with a mean opacity close to 1 emit almost all of their generated particles, while cells with a low opacity end up with sparse particle coverage.

3.2 Particle projection and Image Generation

Particle projection involves two steps. First, the screen space location of the particle needs to be determined. We need the virtual camera parameters and volume transform to achieve this. Second, a color value needs to be assigned to each particle.

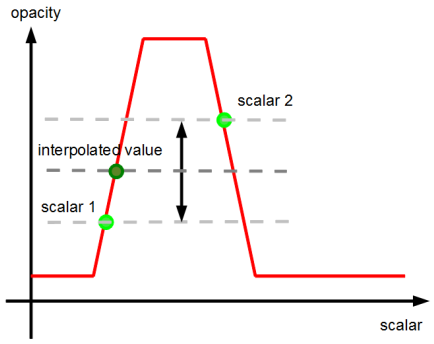
Projection from Object Space to Image Space By using the modelview-projection matrix of the viewing camera, we determine the image-space position of each emitted particle. Further, we calculate its distance to the camera. This involves a simple matrix - vector multiplication. Should two particles hit the same fragment on the image plane, the one closer to the camera is chosen to be displayed. The particle with a bigger distance is discarded. This way, no depth sorting of any kind is necessary before or during rendering. We only need to compare the depth values of subpixels. This approach is comparable to z-buffering. Therefore, it is necessary to create two buffers, one for color, and one for depth.

Transfer function Looking up the corresponding color of a given scalar value in a transfer function is possible at three different stages of our approach. The first possible lookup can happen before projection. This means assigning the corresponding color to the corner points of the cells. To calculate the color of a particle, one needs to interpolate the colors of the corner points. This method is called pre-classification. The next possible lookup may happen during projection of a particle. In post-classification, as opposed to pre-classification, the scalar values of the corner points are interpolated. Assigning a color value to a particle is done by applying the transfer function to the interpolated scalar value. Both pre- and post-classification have the same memory footprint, due to the fact that the RGBA value of a pixel uses the same amount of memory as a floating point scalar value. Also, the computational effort for both methods is roughly the same. Our approach offers one more possibility. One can store the scalar value until subpixel aggregation and perform the lookup for the final interpolated scalar of a pixel.

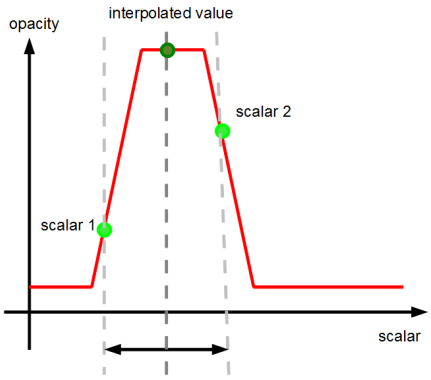
We use post classification per default because of the advantage in interpolation accuracy. Figure 3 illustrates the difference between pre- and post-classification.

3.3 Spatial superimposing

To achieve a higher degree of projection accuracy as well as a translucent appearance of the grid we use spatial superimposing. This means that each pixel is subdivided into several subpixels. The subpixel level l describes the amount of subpixels per pixel, where the number of actual subpixels equals $l * l$. The particles are thereby projected

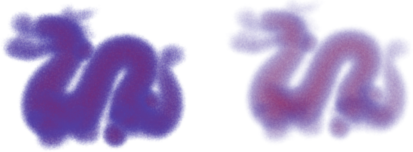


(a) pre-classification



(b) post-classification

Figure 3: The difference in interpolation between different classification strategies. Figure (a) shows pre-classification where final opacity and color values are interpolated. In (b) scalar values are interpolated using post-classification and the application of the transfer function is performed using the interpolated scalar.



(a) l=3 (b) l=6

Figure 4: Stanford Dragon rendered with 60 million particles per frame, subpixel level 3 (a) and subpixel level 6 (b) on a resolution of 1200x800 pixels

onto a subpixel instead of a whole pixel. This increases the amount of particles actually reaching the image plane. Finally, the resulting pixel is calculated by averaging the values of the subpixels.

Translucency Translucency is controlled by two parameters. Firstly, the amount of generated particles influence how opaque the volume seems to be. The more particles are generated, the more subpixels are hit. Secondly, the subpixel level increases or decreases the level of transparency. When there are more subpixels in total, there are also more subpixels, which are not hit by particles. Hence, in the aggregation process we find more empty subpixels, decreasing the mean opacity of a superpixel and making the color appear to be brighter.

Unfortunately, increasing the subpixel level also increases the amount of used memory on the GPU drastically. Therefore, a proper subpixel level considering the tradeoff between accuracy and feasibility needs to be found for each GPU model. Also, the number of particles and the subpixel level need to be balanced for reaching the desired level of transparency and computational performance. This balance needs to be adjusted individually for each graphics card and desired volume translucency.

Particle Depth Enhancement Simple averaging of subpixels results in an unwanted visual effect. The original particle position and thereby the distance to the camera is not taken into account. Thus, averaging treats each subpixel as the same, resulting in an equal visualization of particles disregarding their distance to the camera. The effect is best compared to front face culling in mesh rendering. It might lead to a wrong depth perception while viewing rendered volumes. While it is hardly perceivable on static images, the viewer might become aware of it when rotating or panning the volume. Perceivably, the rendered volume does not respond to transformation as expected. To circumvent this effect, we use the already present depth information of displayed particles.

In detail, we analyze the current depth of each subpixel z_{curr} and record minimum z_{min} and maximum depth z_{max} for a pixel. We then calculate the depth range. Next, we calculate a depth ratio ζ, considering the gap to the maximum value.

$$\zeta = (z_{max} - z_{curr})/(z_{max} - z_{min}) \tag{9}$$

Using ζ as factor for the RGBA values of subpixels, we achieve a linear differentiation of particles respective to their depth values. Particles with a higher distance to the viewer have a smaller impact on final pixels than particles close to the camera.

4 Implementation

We implemented our method using CUDA 3.2 [1], C++ and OpenGL using a NVidia GForce GTX 470 graphics card. We use GLUT to create an OpenGL viewer and to provide necessary camera controls.

4.1 Preprocessing of Datasets

We preprocess the datasets using the Visualization Toolkit [10]. We iterate over all cells and determine their type. If we encounter a non-tetrahedral cell we tetrahedralize the cell if possible. In the next step we convert the VTK Unstructured Grid to a VBO containing a simple data structure. Each tetrahedron consists of four vertices plus their corresponding scalar value. We then map this VBO to the GPU for final processing.

4.2 GPU Preparation

We need to store the transfer function for the scalar value lookup and we also need to transfer the current model-view-projection matrix to the device. Cuda random number generation needs an array of states, which we allocate and setup on the GPU once. Furthermore, we allocate different buffers in the GPU's memory. We create two buffers for storing the projected particles in subpixels, one for color and one for depth. Finally, we use a pixelbuffer for the final image, which we can map to a texture on the screen.

4.3 Cuda Kernels

Projection Kernel The computing grid of this kernel is linear. We use a blocksize of $256 \times 1 \times 1$ and calculate the grid size to be $\#tetras/256 \times 1 \times 1$. Each thread handles one tetrahedron. First we calculate the number of particles to be generated by the current thread. For each particle, we perform the following steps. First, we generate the barycentric parameters according to Section 3.1. Next we calculate the particle's scalar as described in Section 3.1 and perform an opacity-only lookup for this scalar. We test this scalar against a random number, which we generate to determine whether it is projected or not. This process is depicted in the flow chart of Figure 5. If this test succeeds, we calculate the position of the particle and finally project it via multiplication with the model-view-projection matrix. This generates screenspace coordinates, which we look up in our buffer. If the corresponding subpixel is already covered by a particle, we perform a depth check to determine whether we need to overwrite the current subpixel and the depth value accordingly. A visualized flow of this process can be seen in Figure 6.

Superimposing Kernel We define a block size of $16 \times 16 \times 1$ and a grid size of $windowwidth/16 \times windowheight/16 \times 1$. Each thread handles one block of

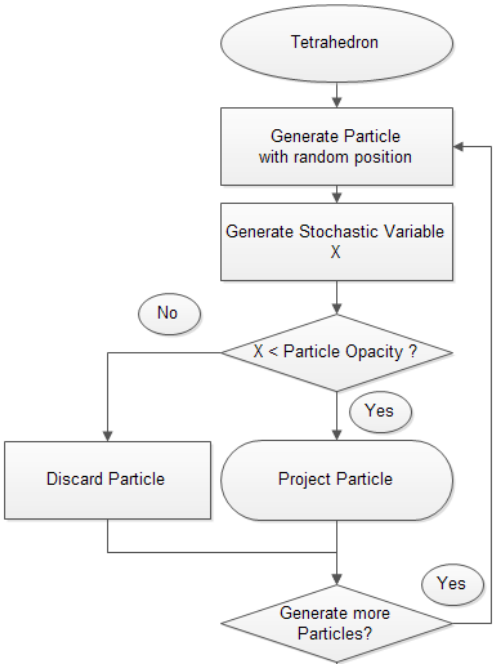


Figure 5: Flow chart depicting particle generation

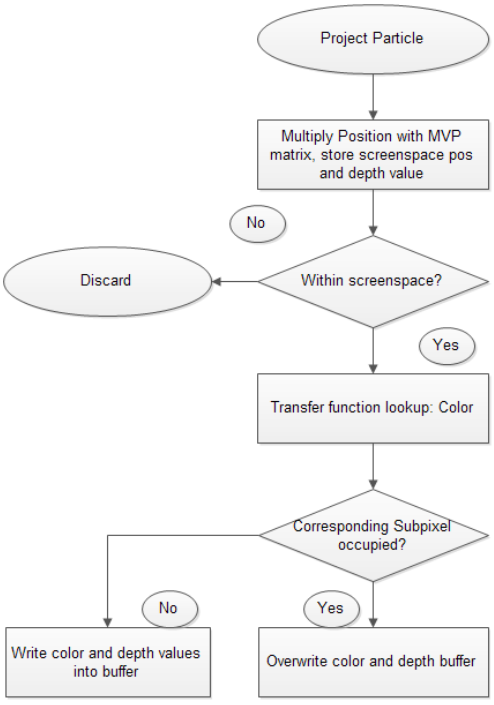


Figure 6: Flow chart depicting particle projection

subpixels corresponding to the number of subpixels per pixel. In a preliminary step we determine the minimum and maximum depth value in the current subpixel block as well as the distance between those extremes. We consecutively look up each subpixel, multiply it by the value described in Section 3.3, and add them up. Finally we divide the summation by the total amount of subpixels per

pixel and store the calculated value in our pixelbuffer.

Gaussian Smoothing As an optional step for improving visual results of our renderer, we post process the images we generate. Using the same computation grid as above, we apply a 3×3 gaussian blur to the pixelbuffer and store the result in the final pixelbuffer, which is displayed. Figure 7 shows a comparison of two images, one of which was smoothed, and the other was not.

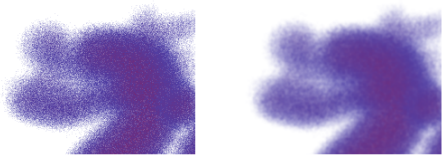


Figure 7: Stanford Dragon with 60 million particles, subpixel level 3. The figure on the left shows the unsmoothed version. The figure on the right contains the same area of the volume, but with a 3x3 gaussian kernel applied to it.

4.4 Displaying Rendered Images

To finally display the rendered image, we use a screen-sized texture quad and map the filled final pixelbuffer to it. As all our data structures are already OpenGL-conform, there is no further processing neccessary.

5 Results

For testing we use several volumetric datasets. One of our main test volumes is the Stanford Dragon volume [18]. We have tetrahedralized a voxel volume to obtain a regular tetrahedron grid. This grid consists of 588245 tetrahedral cells. Another test volume is a completely unstructured grid obtained from a simulated radio frequency ablation in liver tissue as described in [4] and [9]. This test volume consists of 55527 cells.

Figure 4 shows two images of the Stanford Dragon rendered with a maximum amount of 60 million particles on subpixel level 3 and 6. A comparison of those images exposes the importance of balancing the amount of particles with the subpixel level to obtain the desired transparency of the volume.

In Figure 8 we show the decrease in performance with increasing number of particles and increasing subpixel levels tested with the Stanford Dragon volume. From a certain point, neither increasing the subpixel level nor increasing the particles by one step drastically decreases the frame rate. Taking the high amount of cells of this dataset into account, the recorded data show that our approach scales well with a high amount of particles and a high subpixel level.

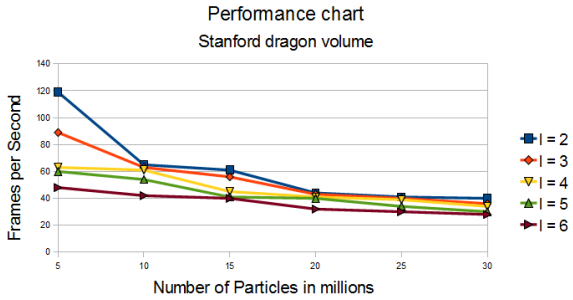


Figure 8: Frames per second for different amounts of particles and subpixel levels while rendering the Stanford Dragon volume. Lines depict the flow of performance for fixed subpixel levels.

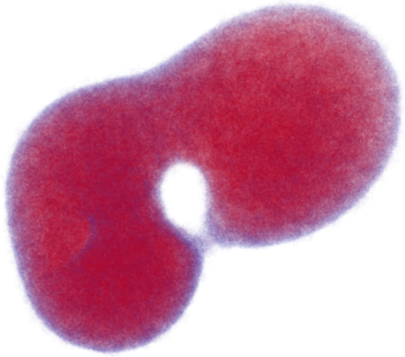


Figure 9: Radio frequency ablation simulation, subpixel level 3 and 6 million particles per frame at 55 fps

Figure 9 shows the radio frequency ablation simulation dataset. The dataset was constructed from a finite element simulation, using an unstructured grid. We have preprocessed this grid to split up non-tetrahedral cells. The volume itself shows probability of cell death during a radio frequency ablation. The saturated, red areas in the center have a high probability of cell death while the blueish border regions are more likely to survive the treatment. Hinted on the left and obvious in the center, the viewer can see veins penetrating the area, working as a heat sink and thereby increasing probability of cell survival. This is depicted in the hazy border regions as those areas have lower opacity, and thereby a lower amount of particles to be emitted.

6 Conclusion

We have shown that our method is able to render millions of particles per second in real-time. This is mainly possible achieved by our particle generation process. We generate the particles per-frame and in real-time, and take care of proper distribution over the volume. Further, our approach offers capabilities to render arbitrary volumetric data structures as most data sets can easily be converted to a tetrahedral structure. The opposite, correct voxelization of unstructured data, is a lot harder to achieve.

In the future, this approach might be expanded to enable rendering multiple volumes. Also, the approach could be modified to distributedly render very high resolution images for large displays. A detailed benchmark test and comparison against other volume rendering approaches is also part of future work.

References

[1] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[2] R. Avila, Taosong H., Lichan H., A. Kaufman, H. Pfister, C. Silva, L. Sobierajski, and S. Wang. VolVis: a diversified volume visualization system. In *Visualization, 1994., Visualization '94, Proceedings., IEEE Conference on*, pages 31 –38, CP3, Oct. 1994.

[3] I. Babuska. Generalized Finite Element Methods : Main Ideas , Results , and Perspective. *Security*, 1(1):67–103, 2004.

[4] T. Bien, G. Rose, and M. Skalej. FEM modeling of radio frequency ablation in the spinal column. In *Biomedical Engineering and Informatics (BMEI), 2010 3rd International Conference on*, volume 5, pages 1867 –1871, oct. 2010.

[5] T.P. Caudell and D.W. Mizell. Augmented reality: an application of heads-up display technology to manual manufacturing processes. In *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, volume ii, pages 659 –669 vol.2, jan 1992.

[6] J. Challinger. Scalable parallel volume raycasting for nonrectilinear computational grids. In *Proceedings of the 1993 symposium on Parallel rendering*, PRS '93, pages 81–88, New York, NY, USA, 1993. ACM.

[7] A.S. Glassner. Generating random points in triangles. In *Graphic Gems*, pages 24 – 28. Academic Press, 1990.

[8] Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, Daniel Weiskopf, and Klaus Engel. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006.

[9] D. Haemmerich, S. Tungjitkusolmun, S.T. Staelin, Jr. Lee, F.T., D.M. Mahvi, and J.G. Webster. Finite-element analysis of hepatic multiple probe radio-frequency ablation. *Biomedical Engineering, IEEE Transactions on*, 49(8):836 –842, Aug. 2002.

[10] Kitware Inc. The Visualization Toolkit. `http://www.vtk.org`, February 2012.

[11] A. Maximo, Marroquim R., and Farias R. Hardware-Assisted Projected Tetrahedra. *Computer Graphics Forum*, 29, Issue 3:903–912, 2010.

[12] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[13] Christian P. Robert and George Casella. *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[14] S. Roettger and T. Ertl. A two-step approach for interactive pre-integrated volume rendering of unstructured grids. In *Volume Visualization and Graphics, 2002. Proceedings. IEEE / ACM SIGGRAPH Symposium on*, pages 23–28, Oct. 2002.

[15] N. Sakamoto, J. Nonaka, K. Koyamada, and S. Tanaka. Particle-based volume rendering. In *Visualization, 2007. APVIS '07. 2007 6th International Asia-Pacific Symposium on*, pages 129 –132, Feb. 2007.

[16] Naohisa Sakamoto, Jorji Nonaka, Koji Koyamada, and Satoshi Tanaka. Volume Rendering Using Tiny Particles. In *Multimedia, 2006. ISM'06. Eighth IEEE International Symposium on*, pages 734 –737, Dec. 2006.

[17] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. In *Proceedings of the 1990 workshop on Volume visualization*, VVS '90, pages 63–70, New York, NY, USA, 1990. ACM.

[18] Stanford University. The Stanford 3D Scanning Repository. `http://graphics.stanford.edu/data/3Dscanrep/`, February 2012.

[19] R. van Pelt, A. Vilanova, and H. van de Wetering. Illustrative Volume Visualization Using GPU-Based Particle Systems. *Visualization and Computer Graphics, IEEE Transactions on*, 16(4):571 –582, July-Aug. 2010.

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

[21] Changgong Zhang, Ping Xi, and Chaoxin Zhang. CUDA-Based Volume Ray-Casting Using Cubic B-spline. In *Virtual Reality and Visualization (ICVRV), 2011 International Conference on*, pages 84 –88, Nov. 2011.