Image-Space GPU Metaballs for Time-Dependent Particle Data Sets.

Christoph Müller
Universität Stuttgart
31 PUBLICATIONS **478** CITATIONS
SEE PROFILE

Sebastian Grottel
Technische Universität Dresden
39 PUBLICATIONS **562** CITATIONS
SEE PROFILE

Thomas Ertl
Universität Stuttgart
769 PUBLICATIONS **14,150** CITATIONS
SEE PROFILE

Image-Space GPU Metaballs for Time-Dependent Particle Data Sets

C. Müller, S. Grottel, T. Ertl

Visualisierungsinstitut der Universität Stuttgart
Email: {mueller,grottel,ertl}@vis.uni-stuttgart.de

Abstract

Molecular dynamics simulations are today a widely used tool in many research fields. Such simulations produce large time-dependent data sets, which need to be interactively visualised allowing efficient exploration. On the other hand, commonly used point-based rendering of the individual particles usually fails to emphasise global contiguous structures like particle clusters. To solve this issue, we want to visualise these data sets using metaballs. Free particles form individual spheres while clustered particles result in larger closed shapes. Using image-space hardware-accelerated techniques provides interactive frame rates and high visual quality. We present two approaches evaluating the metaballs shapes on the graphics hardware. The first approach uses a vicinity data structure stored in graphics memory to evaluate the metaballs shape in a single rendering pass. The second approach uses multiple rendering passes to approximate the metaballs shape on a per pixel basis.

1 Introduction

A large number of technical and scientific problems can nowadays be solved using molecular dynamics simulations. One such example is nucleation, i. e. the state change from a vapour into the liquid phase. This process is found in many physical phenomena, e. g. the formation of atmospheric clouds or the processes inside steam turbines, where a detailed knowledge of the dynamics of condensation processes can help to optimise energy efficiency and avoid problems with droplets of macroscopic size. The liquid phase emerges through spontaneous density fluctuations in the vapour which lead to the formation of molecular clusters, the predecessors of liquid droplets. The left and middle image of figure 1 show such nucleation data sets. For the calculation of key properties like the nucleation rate, it is essential to make use of a meaningful definition of molecular clusters, which currently is a not completely resolved issue.

The visualisation of the simulation results, especially the emphasis of molecular clusters, is therefore essential for analysing simulation results and cluster detection algorithms. Point-based rendering of such data sets is a common practice and delivers fast visualisations of large amounts of particles, but does not convey the detailed shapes of clusters and obscures their real extents — an effect the users do not desire (see figure 1). On the other side, with metaballs [1] a much more suitable technique for solving this issue exists, since they form a closed and compact surface. However, current metaball rendering techniques either require costly isosurface extraction or are based on time-consuming raycasting, which prevents interactive visualisation of time-based data sets.

This paper presents our efforts to enable interactive visualisation of clusters in molecular dynamics data sets as metaballs. We think that image-space approaches are more promising in this context since they do not rely on preprocessing, like isosurface extraction, and hence should permit rendering of time-dependent data sets. Furthermore, image-space techniques, which include raycasting, should achieve an optimal visual quality without a prohibitively fine tessellation of the isosurfaces.

The remainder of this document is structured as follows: Section 2 briefly describes the theoretical background of metaballs and existing rendering algorithms. In section 3, we describe two approaches for implementing metaballs in image-space, one using additional texture memory and one using multiple rendering passes for solving the occlusion problem. Section 4 presents the results of our performance measurements and is followed by some final conclusions.

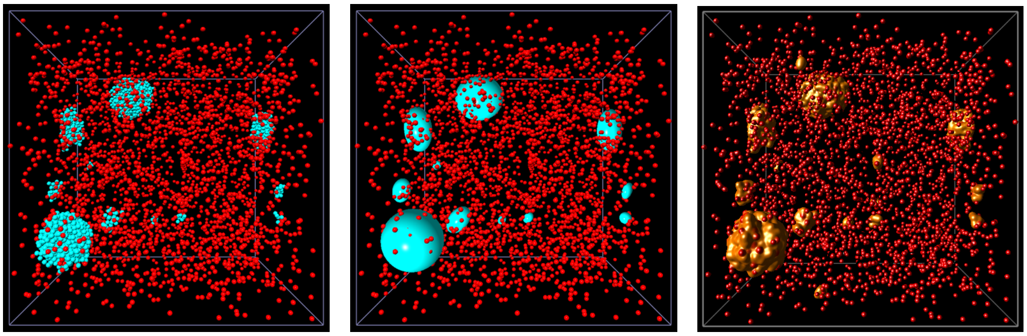
Figure 1: Point-based visualisation of an argon nucleation simulation data set. The left image highlights molecular clusters using different colours, the middle image uses ellipsoids and the right one uses metaballs. The metaball visualisation shows a closed surface of the molecule clusters while not exaggerating their size.

2 Related Work

The metaball technique was introduced by Blinn [1] for displaying molecular models and was originally called *blobs*. The idea is visualising molecules as isosurfaces in the simulation of an electron density field. Blinn uses the exponential function

$$D(r) = \exp(-ar^2)$$

as field function, which is derived from the density function of a hydrogen atom, with r being the distance of the current sampling point from the centre of the current metaball. The sum of the density functions of all atoms parameterised with the distance from the sampling point then yields the value of the density field at this point.

The notion of *metaballs* for the same concept goes back to Nishimura et al. [11][12], who use a piecewise quadratic field function. Several other field functions have been proposed later on, like a degree six polynomial by Wyvill et al. [16] or the degree four polynomial

$$D(r) = \left(1 - \left(\frac{r}{R}\right)^2\right)^2 \tag{1}$$

with R being the radius of the influence sphere of the current metaball by Murakami et al. [9][12]. For a radius $r > R$, $D(r) = 0$ is assumed.

Two typical approaches of displaying metaballs are raycasting the density field and thus rendering the isosurface directly as suggested by Blinn or extracting the surface using the Marching Cubes Algorithm [7] and rendering it like any other mesh.

Using state-of-the-art graphics hardware, Nvidia showed how to use vertex and geometry shaders to evaluate the density field function and generate geometry on the GPU [13]. Kooten et al. [6] employed a point-based method for visualising metaballs on the GPU. Their goal is to render fluid surfaces made up of metaballs, e. g. water in a glass. To achieve this goal, they render a large number of particles which are forced on the implicit surface of the metaballs by velocity constraints and use a spatial hashing method for evaluating the density field on the GPU. Using repulsion forces between the particles, these are uniformly distributed over the surface and thus finally cover the whole metaballs, if their number is sufficiently large.

Earlier, Microsoft showed in their DirectX SDK [8] an image-based approach for achieving a metaball-like effect. They accumulate the density function and the surface normals via additive blending in off screen buffers during multiple rendering passes. As the approach has no depth information, it is limited to metaballs on a single plane.

Our framework for particle data visualisation [3], which the extension for displaying metaballs presented in this paper is intended for, uses glyphs based on the ellipsoid splatting technique presented by Gumhold [4], or more precisely, the approach by Klein et al. [5], which unfolds each glyph from a single point sprite vertex. The algorithm uses e. g. for an ellipsoid the centre point, the three radii and an orientation quaternion to compute the outline of the projected glyph in the vertex shader. The size of the projection determines the size of the point sprite. The fragment shader then raycasts the glyph for each pixel the point sprite generates. This method does not only allow for raycasting spherical and ellipsoidal glyphs, but also more complicated ones like dipoles, arrows and *tubelets* [14].

3 Image-Based Metaball Rendering

The density function we use in our work is a scaled version of the function presented in equation 1:

$$D(r) = \frac{16}{9}\left(1 - \left(\frac{r}{2R}\right)^2\right)^2 \qquad (2)$$

with R being the radius of the sphere forming the metaball, $r \geq 0$, and assuming $D(r) = 0$ for $r > 2R$. The idea behind these factors is to generate a visually coherent impression when switching from raycasted spherical glyphs to metaball rendering or when mixing different glyphs and the metaball rendering. We choose the influence radius to be the doubled particle size of our data sets, and the threshold t for the isosurface in the density field is chosen such that a sphere, which does not form a metaball, looks identical to raycasted sphere glyphs. The scaling of the function with $\frac{16}{9}$ results in the density being one at the distance of the original sphere radius, i. e. $D(R) = 1$. Figure 2 shows a plot of our density function. However, our approaches are not limited to these choices. With minor changes to the used shader programs, it is possible to use any other density function, as long as the function has a finite support, any other influence radius, and any threshold value.

The two approaches presented in sections 3.1 and 3.2 have the common goal to produce a metaball visualisation from spheres, which are solely specified through their centre and radius, in image space without generating geometry. They therefore also share the same problem of occluded fragments, which will be described more detailed in the following section.

Our first idea to overcome this problem was making the missing information available to the fragment shader via a *Vicinity Texture*. The metaballs then can be rendered as point sprites in a single pass without processing the whole image area but at the expense of an increased number of expensive texture accesses. This approach is closely related to the glyph raycasting in our framework and described in section 3.1.

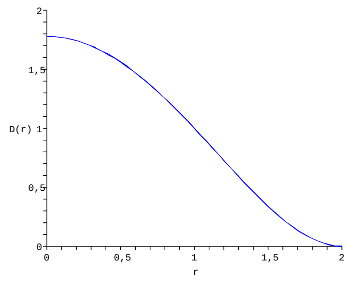

Figure 2: The density function from equation 2 with $R = 1$.

Our second approach, which is elaborated in section 3.2, goes quite in the opposite direction: instead of generating fewer but more expensive pixels, it constructs the metaballs from a lot of relatively cheap pixels in several rendering passes. Reminding of depth peeling [2], the final image results from a viewport aligned imaginary plane being moved through the data set. In each pass, the density function is evaluated for each pixel and the plane is moved by a distance value provided by an oracle. If these steps are adequately long and enough of them have been made, the surface of all metaballs should have been found.

3.1 Vicinity Texture

The main problem when trying to evaluate 3D metaballs in image space is occlusion, which prevents an accumulation of the density function by simply rendering the projection of the influence spheres of the particles. E. g. even if one knew in advance that $P1$ in figure 3 will not contribute to a metaball and therefore only rendered the surely opaque sphere with radius R instead of the whole influence sphere with radius $2R$, none of the fragments generated by $P3$ would be visible. Consequently, when accumulating the density within the projected influence sphere of $P2$, the contribution of $P3$ and therefore also the dotted metaball surface connecting $P2$ and $P3$ would be missing.

One would have to know for every fragment generated by an influence sphere which other spheres contribute to it, i. e. this information would have to be available on the graphics card. Kooten et al. [6] use a reversed version of the spatial hash table pre-

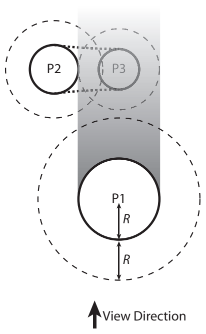

Figure 3: Occluded fragments hamper the image space computation of metaballs: the dashed influence sphere of $P3$ is completely occluded by $P1$. Therefore, it is not possible to accumulate the density function correctly in image space. As the contribution of $P3$ is completely missing, the dotted metaball surface cannot be found and $P2$ appears as normal sphere.

sented by Teschner et al. [15] to obtain for a certain area of the screen all relevant particles. Depending on the size of the grid subdividing the screen space, this hash table can become quite big, and as we intend to use the metaball visualisation in combination with glyphs raycasted on the GPU as described by Klein et al. [5] we chose to attach the vicinity information not to the screen position but to the vertices defining the object-space centre of the spheres. This has also the advantage that our map is view-independent and must only be updated, if the position of the vertices changes. Thus, we construct in a pre-processing step a texture which holds for each sphere the object-space position and the radius of all other spheres which are near enough to contribute to the density field. Before each set of influencing spheres, an extra entry holding the number of positions to follow is added. The coordinates of this counter are set as texture coordinates of the vertex. All spheres which do not have neighbours with which they potentially form a metaball are omitted in order not to waste texture memory.

Having the above-mentioned lookup table for the influencing spheres, the rendering of the metaballs happens in a single pass: the centre positions of the spheres are rendered as point sprites and the vertex shader adjusts the screen space size of the point to hold all the pixels of the projected sphere. For that, the desired object-space sphere radius is passed as homogenous coordinate of the vertex. The sign of the homogenous coordinate is additionally used to determine, whether the current sphere potentially forms a metaball. If it can form a metaball, its radius is doubled in order to make the sphere enclose the whole space where our density function (equation 2) does not vanish. In the fragment shader, the sphere is then raycasted, and as long as it surely cannot be part of a metaball the intersection point between the viewing ray and the sphere just has to be lit to complete the pixel [5].

If, however, the current sphere can form a metaball, we must determine whether it actually does so, and where the threshold value t is reached for the first time. For these purposes, we sample the density field within the influence sphere, whose boundary we just have computed, while advancing on the viewing ray until the end of the influence sphere. At each sampling point, the density function is evaluated on-the-fly for the current sphere and the ones in their neighbour list. If the sum of the densities, which is the final density at the sampling point, is sufficiently close to the isovalue t, the sampling ends.

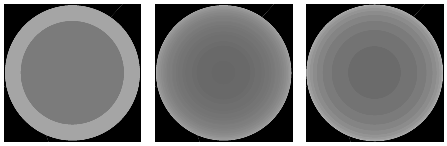

Figure 4: Influence of the recursive refinement of the isosurface. The colour denotes the length of the ray from the surface of the influence sphere to the isosurface in the density field. The leftmost image shows the result for only ten sampling points, the middle one for ten sampling points and four additional refinement steps, and the rightmost one for 50 sampling points.

It requires, however, a prohibitively large number of samples per pixel in order to find a sufficiently good isosurface in the density field. But the number of required samples can be effectively limited if the sampling is stopped once the density is above the threshold for the first time and the desired point on the isosurface is then searched by bisecting the previous sampling step recursively. Figure 4 shows this effect for four recursive refinement steps.

Once the isosurface of the metaball in the density field has been found, the OpenGL-conform depth of the fragment can be computed like for a normal sphere or any other glyph. For lighting the metaball surface, the normals are approximated as the sum of the normals of the contributing spheres weighted with the respective density contribution [8]. Since the accumulated density on the surface results to one, the weighted summation of normal vectors is equivalent to the use of barycentric coordinates and thus a legitimate interpolation.

3.2 The Walking Depth Plane

As we found that the texture lookup in the vicinity texture poses heavy load on the graphics card (see section 4), we tried an alternative approach without using a texture-based data structure. The goal was to remove the texture lookup as bottleneck and additionally to avoid restraints on data set sizes due to the available texture memory. The main idea of this approach is to use multiple rendering passes to approximate the correct isosurface through a moving plane of depth values.

Two frame buffer objects with 16 bit float RGB channels are used to store the needed data. The first buffer, called λ-buffer from now on, is used to store the depth of the pixel in image space, which approximates the targeted isosurface in means of the distance from position of the camera in world space coordinates. In a second channel of this buffer the maximum distance value for the pixel is stored as information for the termination criterion. The second buffer — we will refer to this one as density buffer — is used to evaluate the density field at the depths provided by the λ-buffer. Using these two buffers, the algorithm works as follows:

3.2.1 Initialising the λ-Buffer

The first rendering pass (Step 1 in figure 5) is used to initialise the λ-buffer. The starting and maximum depth values over all spheres for each pixel are calculated by raycasting the influence spheres of each particle. Depending on whether the starting or the maximum values are calculated, the front or back side of the spherical glyphs are raycasted. The minimum or maximum over all glyphs is determined using OpenGL depth tests. The pixels which are not set by any fragment of these glyphs are initialised using the clear colour value. Since these pixels will never be part of a metaball, their values must only trigger the loop termination criterion and have not to be comparable to real λ values calculated using the initialisation sphere glyphs.

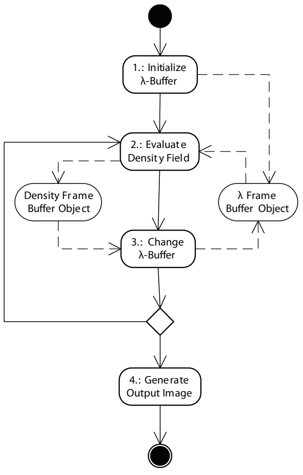
1.: Initialize λ-Buffer
2.: Evaluate Density Field
Density Frame Buffer Object
λ Frame Buffer Object
3.: Change λ-Buffer
4.: Generate Output Image

Figure 5: The different rendering passes of the *Walking Depth Plane* approach, together with the two frame buffer objects involved. The solid lines represent the control flow, while the dashed lines represent access to the frame buffer objects.

3.2.2 Evaluation of the Depth Field

After the initialisation step, the approximation of the metaball surface is done by repeating steps 2 and 3 (see figure 5). This loop and its termination are controlled by the CPU.

The evaluation of the density field in step 2 is very similar to the initialisation pass. The individual spheres are uploaded as points with the additional information of the influence radius of the density function. But instead of raycasting an implicit surface of the sphere, the density function for the sphere is evaluated at the position calculated from the viewing ray of the given pixel and the depth value stored in the λ-buffer. All density portions of all spheres are accumulated by performing additive blending on the float frame buffer the values are written to. To improve the upcoming update of the λ-buffer the density field can be evaluated at multiple depths at one time by summing the density at a specific depth in a separate channel of the frame buffer object. Using a single RGB colour attachment allows for evaluating the density field at three positions at a time, which allow for a more substantiated guess of the next sampling position. A similar use of colour channels was presented in [10] for performing RBF-based volume rendering through splatting on up to four slices in one pass.

3.2.3 Moving the Depth Plane

The third step is to change the values of the λ-buffer and thus to move the surface described by these values closer to the targeted isosurface. Since the values are stored in a float colour attachment, they can be increased and decreased using alpha blending and fragments with a colour value describing the change step. It is also possible to use two λ-buffers used alternately. This way the calculation of the λ values for the next pass could not only consider the evaluated depth value, but also the current λ values. However, since we wanted to minimise the use of textures, we chose to use a rather simple oracle for computing the step size and direction from only the last λ value.

The direction of the step is directly given by the density value of the current pixel. If the density is below the threshold, the step must go forward into the viewing plane, and if the density is above the threshold, we already missed the isosurface and must step back. While this is intuitive, choosing a step size is not. The radius of the smallest sphere can be used as maximum step size. It is then still possible to miss a rather thin connection between spheres, but this is an intrinsic problem of any sampling. However, in all of our data sets this problem does not occur unless rendering time-dependent interpolated data sets with changing molecule sizes.

To approximate the targeted isosurface with sufficient precision, the step size must be decreased as the density reaches the threshold value. This decrease is controlled by an oracle. Since the density is a summation from several density functions, it is not possible to precisely determine the required step size. Therefore, we use two simple heuristics as our oracle. We can use the fact that in our molecular dynamics data sets the spherical particles only interpenetrate each other a little to our advantage here. Hence, the summed density functions characteristics does not differ dramatically from the individual function. As approximation to these characteristics we used quadratic attenuation for steps into the viewing plane and linear attenuation for steps out of the viewing plane: the step size $|\Delta|$ is

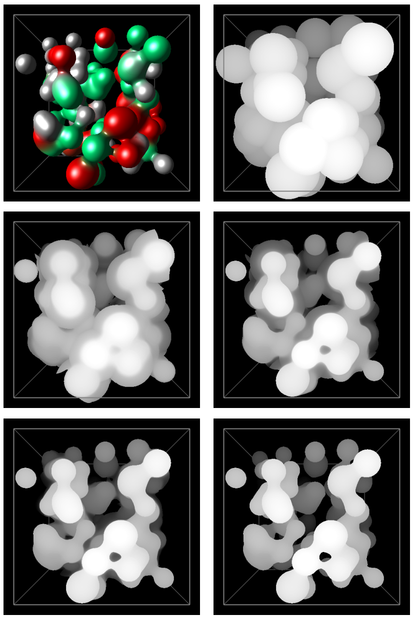
Figure 6: Metaball isosurface approximation after different number of iterations. Upper left image shows final output image of the *Walking Depth Plane* approach. The other images show the content of the λ-buffer after different numbers of iterations (from upper right to lower left): directly after initialisation, after 5, 10, 15, and 31 (final result) iterations.

$$|\Delta| = \begin{cases} \Delta_{max}(1 - \tilde{D}(\vec{x})^2) & \text{for } \tilde{D}(\vec{x}) < t - \epsilon \\ \frac{1}{2}\Delta_{max}(\tilde{D}(\vec{x}) - 1) & \text{for } \tilde{D}(\vec{x}) > t + \epsilon \\ 0 & \text{otherwise} \end{cases}$$

with Δ_{max} being the maximum step size, t the threshold, ϵ the approximation tolerance, and $\tilde{D}(\vec{x})$ the summed density over all metaballs in the data set at the position $\vec{x}$. Experimental results showed that this oracle is quite effective. Figure 6 shows the evolution of the λ values using the oracle with a data set consisting of rather huge metaballs formed by several individual spheres, making it hard to predict the correct step size as already mentioned above.

3.2.4 Terminating the Loop

The last issue to be solved is to define and to evaluate the criteria for terminating the iteration loop. As the maximum step size is known, the required iterations can be computed using this value and the size of the bounding box. This, of course, is only true as long as the oracle does not decrease the step size adaptively. Therefore, the maximum number of iterations must be increased, e. g. by a factor of two. For the data set shown in figure 6, the calculated value is 16 resulting in a maximum number of iterations of 32, which is a very good prediction. However, if the sphere radii are quite small, as in most of the data sets we present, the maximum number of iterations gets overestimated. An overall iteration maximum is used to prevent the application from visually freezing. As second mechanism ensuring the responsiveness of the application, a maximum execution time for the iteration loop is used: a minimum frame rate can be defined, which will be reached under all circumstances by early termination of the loop at the cost of worse approximation of the isosurface in the background of the data set. In our test runs we used a minimum frame rate of 1 FPS.

The last termination criterion of our software is the approximation of the isosurface, which is directly given by the density value calculated in step 2. If this value is sufficiently close to the threshold, the pixel is marked as finished, by setting the value of the depth buffer of both frame buffer objects to zero, which is done in one pass since both frame buffer objects share the same depth attachment. If a given percentage of all pixels is marked as finished, the iteration loop is terminated. Figure 7 shows such terminated pixels after a few iterations. The straight forward approach to evaluate this criterion would be using occlusion queries and count the created fragments relative to the size of the viewport. However, this is not possible on many systems, including all computers we tested our software on, because occlusion queries seam not to count fragments generated into a frame buffer object. As fall back solutions, one could count all texels with a value of zero in an additional rendering step or reading the whole image back and count them on the CPU. However, the overhead for both solutions is so big that it actually is the best idea to abandon this criterion and to rely on the other ones, which perform sufficiently well.

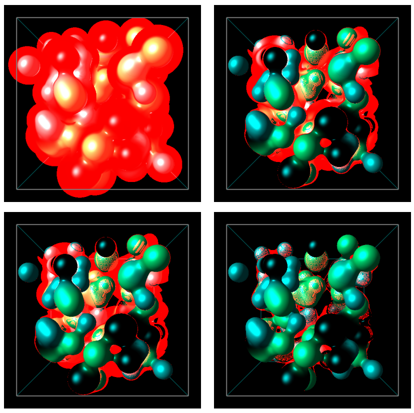
Figure 7: The metaball of the same data set as shown in figure 6 after 5, 10, 20, and 30 iterations (from upper left to lower right). The red colour channel encodes the finished pixels black (showing the isosurface in cyan) and unfinished pixels red.

3.2.5 Shading

The final output image is generated by deferred shading in step 4. After the iteration loop terminated, the λ-buffer holds the approximated distance values of all required points on the targeted isosurface. In an additional rendering pass, the density field is again evaluated at the positions stored in the λ-buffer for computing the surface normal vectors and the colours similarly as described in section 3.1. A final rendering into the real window's frame buffer uses the values from the frame buffer objects to write fragments with OpenGL-conform depth and a colour calculated by a phong lighting using the colours and normal vectors.

4 Results

As already mentioned, our goal was to interactively visualise molecular dynamics data sets with our metaball methods. Tables 1 and 3 show performance measurements for six different data sets. The *Argon* data set (figure 1) is a nucleation simulation of argon with 5000 molecules. The *Ethane* data set (figure 9) — the largest in our tests — consists of 25000 molecules and shows a nucleation process in a supersaturated configuration. Renderings of these two data sets using the *Walking Depth*

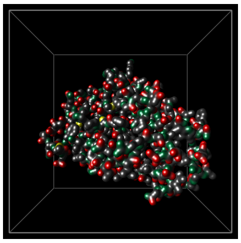

Figure 8: Disulfide Bond Formation Protein from the Protein Data Base consisting of 1454 particles.

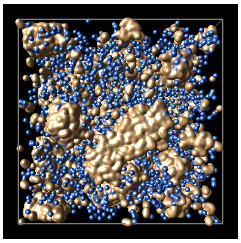

Figure 9: A nucleation simulation of supersaturated ethane consisting of 25000 elements.

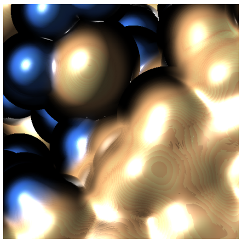

Figure 10: Undersampling artifacts when rendering the ethane data set with the *Moving Depth Plane* and only 100 steps.

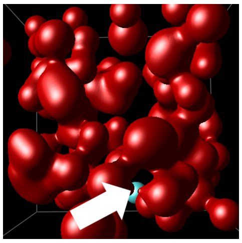

Figure 11: Missed surface parts (arrow) due to undersampling by the *Vicinity Texture* approach.

Plane approach are shown in the video accompanying this paper, captured in real time from the desktop. The data sets three to five, *Sim-1* (figure 12, bottom left image), *Sim-2* (figure 12, top images) and *Sim-3* (figures 6, 7 and 11), are generated test data sets used while developing our software. They were created using a random placement, only controlling that the overlapping of spheres is limited. The sixth data set, *1A2J* (figure 8), is a Disulfide Bond Formation Protein imported from the Protein Data Base as example of the applicability of our method to other kinds of data sets.

Table 1: Rendering Performance of *Vicinity Texture* implementation. The processing time specifies the time needed to build up the required data structure texture (view-independent) using a plane-sweep algorithm.

Data Set	# of Spheres	Processing Time	FPS
Argon	5000	113 ms	0.1
Ethane	25000	2568 ms	0.2
Sim-1	100	<1 ms	61.0
Sim-2	500	1 ms	4.0
Sim-3	100	<1 ms	4.0
1A2J	1454	12 ms	0.14

Table 1 shows timing results for the *Vicinity Texture* implementation. It is striking that the approach shows an extremely poor performance for data sets with large metaball clusterings as our real-world data sets are. We attribute this to the tremendous number of texture fetches required during the on-the-fly evaluation of the density function. Especially noticeable is the fact that the frame rate for the *Sim-1* data set is 15 times higher than for the *Sim-3* data set, although both consist of 100 spheres. Table 2 points out, why: it shows, how many spheres are potentially part of a metaball, i. e. have a non-empty vicinity groups, and how many are normal spheres. As for the *Sim-3* data set nearly all spheres are potential metaballs, the number of texture accesses is much higher than for *Sim-1*. The most extreme data set is the 1A2J data set, which

Table 2: Number of potential metaball members in the different data sets: Non-empty groups are the spheres which have at least one neighbour which is close enough to form a metaball with. Spheres which can never form a metaball are counted as empty groups. The last column shows the minimum, average and maximum vicinity group size in the data set.

Data Set	Non-empty Groups	Empty Groups	Min/Avg/ Max Size
Argon	3684	1316	1/39.3/125
Ethane	24838	162	1/85.2/178
Sim-1	40	60	1/1.7/4
Sim-2	305	195	1/1.6/7
Sim-3	99	1	2/8.0/17
1A2J	1454	0	2/6.9/13

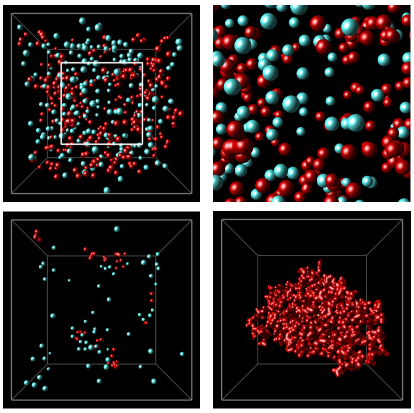

Figure 12: Results of the potential metaball detection during the preprocessing: metaballs and potential metaballs are coloured red, normal spheres cyan. The upper left image shows classification for the 500-element Sim-2 data set, the one right of that shows a close up view of the center area. One can see that actually only very few metaballs are generated. The lower row shows the Sim-1 and the 1A2J data set. In the last-mentioned one, all spheres form one large metaball.

forms one large metaball and therefore has no normal sphere (Figure 12, bottom left image).

One interesting fact about the *Vicinity Texture* approach is that it scales quite well with the image resolution. The 1600×1200 rendering of the *Sim-1* data set reaches the same frame rate as on a 512^2 viewport. We attribute this to the fact that the texture fetching capability of the graphics hardware is not completely utilised for the small number of fragments generated in the small viewport. For the *Sim-2* data set the frame rate drops from 4.0 to 2.0 FPS, because the number of filled pixels, which cause time-consuming texture fetches, increases while texture access posed heavy load on the GPU already for the small picture.

Table 3 shows performance values for the *Walking Depth Plane* approach. Since the approach does not only depend on the complexity of the data set, but also on the maximum number of iterations, we provide rendering performance values for all six data sets using exactly 100 iterations. Increasing or decreasing the maximum number of iterations intuitively decreases or increases the frame rates. No

Table 3: Rendering Performance of *Walking Depth Plane* Implementation for different maximum iteration counts.

Data Set	# of Spheres	Maximum # of Iterations	FPS
Argon	5000	100	13
Ethane	25000	100	5
Sim-1	100	100	61
Sim-2	500	100	15
Sim-2	500	80	19
Sim-3	100	100	15
Sim-3	100	31	61
1A2J	1454	100	21
1A2J	1454	250	11

rendering, except for the rendering of *1A2J* with 100 iterations, created any visual artifacts due to insufficient number of iterations (see figure 10). The *Vicinity Texture* approach can also suffer from typical undersampling artifacts like holes the in metaball surface and from jittering of the surface normals (see figure 11) as we often can afford only very few sampling steps per pixel.

We ran our performance tests on an Intel Core2 Duo 6600 processor with 2.40 GHz, 2 GB memory, and an NVidia GeForce 8800 GTX graphics card with 768 MB graphics memory. The viewport size was 512^2 for all tests.

5 Conclusions

In this paper, we showed how to interactively render metaballs from large particle data sets. We tried two different techniques, of which the first one turned out to perform disappointingly with real-world data sets. The tremendous number of texture accesses required by this approach simply poses a too heavy load on the graphics card, but the evaluation of the density function only for fragments generated by the influence spheres limits the impact of the viewport size after all. As the current linear layout of the vicinity texture is probably unfavourable with regard to caching, it should be investigated whether spatial grouping of the entries can improve the rendering performance.

We described a second approach using multiple rendering passes to approximate the surfaces of the metaballs. Its rendering performance is quite good on state-of-the-art GPUs, and as it does not require any pre-processing it is even possible to visualise time-dependent data sets on the fly. However, the rendering speed is tightly connected to the viewport size and the method may generate visual artifacts if the number of iterations is insufficient. The current termination criteria and estimation of required iterations cannot effectively prevent these if the size of a molecule changes over time and falls below the pre-computed maximum step size.

Another problem is noticeable in figure 7 when comparing the upper right and lower left images. Although ten iterations lie between these images, the differences are almost indiscernible. We attribute this to the fact that the empty space between the spheres cannot efficiently be skipped without additional effort. Therefore, empty-space-skipping is a feature we want to implement in the future.

We also hope that this and further optimisations can improve the rendering performance to interactively handle data sets consisting of hundred thousands of particles, which are not uncommon in the application area of our software, and to support larger output sizes.

6 Acknowledgements

This work is partially funded by the state of Baden-Württemberg in context of the BW-FIT project *Interaktive Visualisierung für Gigapixel Displays* and by Deutsche Forschungsgemeinschaft (DFG) as part of SFB 716. We want to thank Guido Reina for inspiring discussions on the different approaches.

References

[1] J. F. Blinn. A Generalization of Algebraic Surface Drawing. In *ACM Transactions on Graphics*, 1(3): 235–256, 1982.

[2] C. Everitt. Interactive Order-Independent Transparency. Nvidia Technical Report.

[3] S. Grottel, G. Reina, J. Vrabec, and T. Ertl. Visual Verification and Analysis of Cluster Detection for Molecular Dynamics. In *Proceedings of IEEE Visualization*, 2007, to appear.

[4] S. Gumhold. Splatting Illuminated Ellipsoids with Depth Correction. In *Proceedings of Vision, Modeling, and Visualization 2003*, 245–252, 2003.

[5] T. Klein and T. Ertl. Illustrating Magnetic Field Lines using a Discrete Particle Model. In *Proceedings of Vision, Modeling, and Visualization 2004*, 387–394, 2004.

[6] K. van Kooten, G. van den Bergen and A. Telea. Point-Based Visualization of Metaballs on a GPU. In *GPU Gems 3*. Addison-Wesley, 2007, to appear.

[7] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *International Conference on Computer Graphics and Interactive Techniques*, 163–169, 1987.

[8] Microsoft. *DirectX SDK*. http://msdn.microsoft.com/directx

[9] S. Murakami and H. Ichihara. On a 3D Display Method by Metaball Technique. In *Electronics Communication Conference*, 1607–1615, 1987.

[10] N. Neophytou, K. Mueller. GPU accelerated image aligned splatting In *Volume Graphics*, 197–242, 2005.

[11] H. Nishimura, M. Hirai, T. Kawai, T. Kawata, I. Shirakawa and K. Omura. Object Modeling by Distribution Function and a Method of Image Generation. In *Electonics Communication Conference*, 718–725, 1985.

[12] T. Nishita and E. Nakamae. A Method for Displaying Metaballs by using Bézier Clipping. In *Computer Graphics Forum*, 13(3): 271–280, 1994.

[13] Nvidia. *Direct3D SDK 10 Code Samples*. http://developer.download.nvidia.com/SDK/10/direct3d/samples.html

[14] G. Reina, K. Bidmon, F. Enders, P. Hastreiter, and T. Ertl. GPU-Based Hyperstreamlines for Diffusion Tensor Imaging. In *Proceedings of EUROGRAPHICS - IEEE VGTC Symposium on Visualization 2006*, 35–42, 2006.

[15] M. Teschner, B. Heidelberger, M. Müller, D. Pomeranets and M. Gross. Optimized Spatial Hashing for Collision Detection of Deformable Objects. In *Proceedings of Vision, Modeling, and Visualization 2003*, 47–54, 2003.

[16] B. Wyvill, C. McPheeters and G. Wyvill. Data structure for soft objects. In *The Visual Computer*, 2(4): 227–234, 1986.