Vision, Modeling, and Visualization (2017)
M. Hullin, R. Klein, T. Schultz, A. Yao (Eds.)

Interactive Visualization of Gaps and Overlaps for Large and Dynamic Sphere Packings

Feng Gu[1] Zhixing Yang[2] Michael Kolonko[2] Thorsten Grosch[1]

[1] Department of Informatics, TU Clausthal, Germany
[2] Institute of Applied Stochastics and Operations Research, TU Clausthal, Germany

Abstract

To gain insight into many properties of granular matter, a particle packing can be simulated. For a dry particle mixture, collective rearrangement is often used as an iterative process to place the particles. In this paper, we present a new visualization technique to judge the quality of a collective rearrangement simulation of many spheres with a given particle size distribution. In addition to a visualization of the spheres themselves, we directly visualize the gaps and overlaps of the spheres in each iteration. This allows to see the regions where the simulation is not yet converged as well as the free spaces where spheres can still move into. Our method supports millions of spheres at interactive to real-time frame rates, allowing the user to inspect the sphere packing during the simulation. We demonstrate that this type of visualization better shows the structure of the current sphere arrangement than standard techniques like 2D clipping planes and therefore serves as a visual feedback to support the development of the packing simulation.

CCS Concepts

•***Computing methodologies*** → *Scientific visualization; Rendering;*

1. Introduction

Particle packings form the basis of many materials from different fields like concrete, pills and tablets for medical purposes or powders for 3D printing. Important properties of the final material are determined by geometrical properties of the dry particle packing. Simulating and inspecting these packings may therefore help to develop materials with a particular desirable property.

If we approximate particles by spheres, as it is quite common in material sciences (see e.g. [Tor06]), then the mixture is determined by its *particle size distribution* (PSD) that describes the percentage of particles for each radius.

There are several ways to simulate a packing (see e.g. [BBS*02]). If the aim is to obtain a very dense random packing, the so-called *collective rearrangement* (CR) algorithms seem to be superior. Here, a sample of spheres from the PSD is placed randomly in a container which at the beginning is chosen so small that each of the spheres must overlap with others. A repulsion between the spheres is simulated and the container is enlarged stepwise until a non-overlapping placement is reached.

As this algorithm does not aim at simulating the true physical forces that generate a packing, it is of great importance to inspect the final packing as well as its generation process to judge the quality of the simulation. It may e.g. happen, that some overlaps between spheres remain or that there are unrealistic holes in the packing. If the packing simulates a *foam*, then the spheres represent gas bubbles and the main interest is in the shape of the interstices.

Therefore the development of a flexible *visualization tool* for the simulation became necessary. It should allow the material scientist to inspect the final packing for its properties, but it should also help the developer to check the correctness of the complex simulation algorithms and the impact of their parameters.

More precisely, it should allow to navigate through the packing, select spheres, inspect its overlaps and visualize the free space. This should not only be possible for the final static packing but also during its generation to see the repulsion between the spheres and the gradual vanishing of the overlaps. Realistic simulation of e.g. concrete mixture needs huge samples of particles (see [KRW10]). An additional challenge is the large variation of sphere sizes needed in one sample for a realistic simulation of e.g. concrete mixtures. They comprise spheres with diameters from 0.1 micrometers up to centimeters. Today, highly parallel simulations on the GPU allow one CR iteration with millions of spheres within milliseconds ([YGGK17]).

Although sophisticated methods exist to visualize the spatial placement of particles, simply drawing spheres in 3D does not show where the *remaining overlaps* or the *existing gaps* between the spheres are (see Figure 1). We therefore develop a visualization for collective rearrangement of spheres that allows a direct *rendering of gaps and overlaps*. Our method runs entirely on the GPU

DOI: 10.2312/vmv.20171265

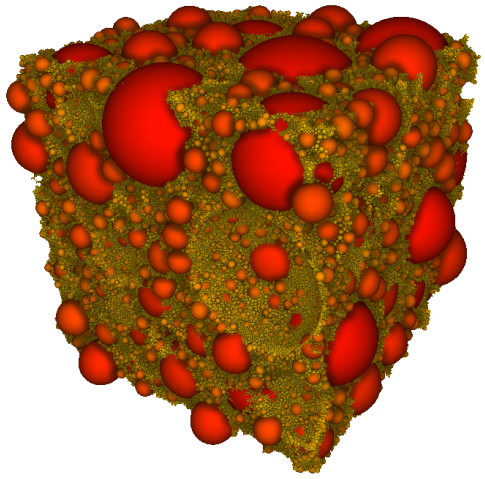

Figure 1: *A sphere packing consisting of one million spheres. Large spheres are drawn in red, small spheres are drawn in yellow. The periodic boundary shows the space that spheres from the opposite side occupy. Due to the unfinished collective rearrangement process, some of the spheres are still overlapping, which is hard to see in a standard visualization.*

and supports large and dynamic sphere packings at interactive to real-time frame rates. The user can interactively inspect the current simulation and see where spheres still overlap, as well as the remaining free space between them. We demonstrate that such a visualization is more helpful than standard techniques like drawing the spheres in combination with 2D clipping planes.

These visualization tools were developed in close cooperation with a GPU-based simulation tool which is described in more detail in [YGGK17]. Similar to the former sequential simulation programs that e.g. utilized loose octrees to store the sphere locations (see [KRW10], [RK11]), the GPU-based simulation also makes use of concepts from visualization so that both, simulation and visualization, use identical core data structures that can easily be exchanged between both parts of the program system.

2. Previous Work

To achieve fast rendering of spheres, billboards are typically used in combination with an intersection test per pixel [MTC06]. There are several projects that draw millions or even billions of spheres at interactive to real-time frame rates, including the whole cell visualizations [FKE13], MegaMol [GKM*15] and molecular reactions [MPSV14]. A recent overview can be found in [KKF*16].

Several rendering techniques exist to emphasize the spatial arrangement of the spheres, including ambient occlusion, depth-aware silhouettes and depth darkening [MTC06]. In case of a large number of spheres, image-space methods can be used since the rendering time only depends on the number of pixels. By inspecting neighboring pixels in a geometry buffer, contact shadows and indirect illumination can be computed within a small region [RGS09].

Our work uses GPU linked lists introduced in [YHGT10] to store information from multiple spheres for each pixel. This allows to draw both overlaps and gaps between the spheres at pixel precision. Our work bears similarities with CSG operations that are computed in image space [KD04]. The idea of using linked lists for CSG operations is described in [RFV13], and [RN14] describes a possible implementation, which is restricted to two objects and requires two render passes for list generation. In contrast, our method deals with an arbitrary number of spheres, fills the lists in a single render pass and reduces the memory requirements for each list element.

3. Collective Rearrangement Simulation

To determine the spatial placement of the spheres, we use a parallel GPU version of the algorithm described in [RK09, KRW10]. Given an arbitrary particle size distribution, we start with random sphere positions, and then use collective rearrangement to remove the overlaps. In this iterative process, overlapping spheres push each other away by a slight amount in each iteration. To cope with a large number of spheres, the spheres are inserted in a loose octree according to their size. In this way, possible candidates for collision are detected quickly. Our simulation uses a cubic container in combination with different border types. Beside a hard border that prevents the spheres from leaving the container, we also support a *periodic boundary* that allows a *tiling* of the sphere packing where spheres can *wrap-around* at each border. In this case, spheres intersecting the border are virtually duplicated for collision checks at the opposite border. Our method requires roughly 30 ms for one CR iteration with one million spheres and achieves good correspondence with measured densities of real materials. The details will be available in [YGGK17], the simulation itself is not a contribution of this paper.

4. Direct Sphere Visualization

Our method requires a visualization of spheres with pixel precision to correctly detect the existing gaps and overlaps between the spheres using a perspective projection. We start by drawing a point for each sphere (VBO with position, color and radius), and use a geometry shader to generate a view-aligned quad for each point. To conservatively rasterize the sphere region, the quad is oriented orthogonal to the sphere center direction with a size adjusted to the sphere radius. Finally, a fragment shader is used to compute the intersection of each pixel ray with the sphere. In case of an intersection, the intersection point, the intersection normal and the sphere color are drawn into a geometry buffer. Otherwise, the fragment is discarded. Afterwards, we use deferred shading and illuminate all visible fragments based on the information in the geometry buffer (position, color, normal). Finally, we use screen-space ambient occlusion [RGS09] to display contact shadows between nearby spheres to intensify the visual impression of the placement.

5. Visualization of Overlaps and Gaps

Although our visualization of spheres gives a good impression of the spatial arrangement of the spheres, it is hard to judge where the remaining overlaps and free spaces are. While this is easy to see in 2D, it is practically invisible in 3D. To better evaluate the quality of the collective rearrangement, we developed a direct visualization

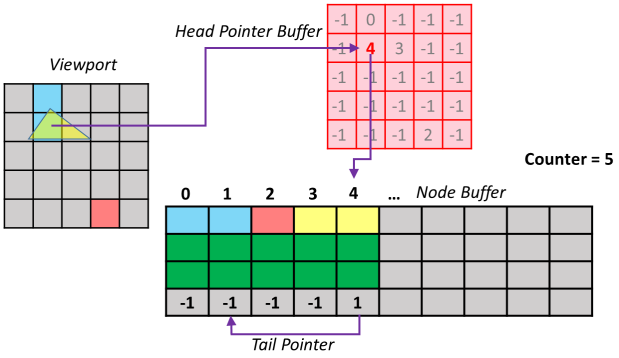

Figure 2: *Calculations for a GPU linked list (based on [YHGT10]). Each column in the node buffer stands for a node. Information needed later such as index can be stored in the green parts for each node.*

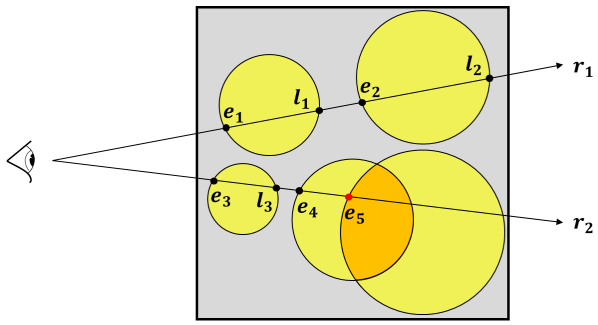

Figure 3: *Rendering overlaps: For non-overlapping spheres, we have alternating enter and leave points (ray r_1). In case of an overlap, we get two successive enter points (e_4 and e_5 on ray r_2). The second enter point (e_5) is drawn. The overlap part is marked in orange.*

of gaps and overlaps in 3D. Since we work with a large number of potentially small spheres we decided to create this visualization directly in *screen space* with pixel precision.

GPU Linked Lists Our work is based on the idea of GPU linked lists introduced in [YHGT10], which provides a method to dynamically construct highly concurrent linked lists on modern graphics processors. To construct a GPU linked list, two buffers are needed: One large *node buffer* which contains nodes of the linked list and another screen-sized *head pointer buffer* to store head pointers, each pointing to the start of a linked list in the node buffer, for each pixel. Figure 2 shows a small example: Here the pixel which has the value 4 on its head pointer buffer has its node information stored on the position 4 in the node buffer. This node contains a tail pointer that refers to the position of the last node that its pixel owns, which is 1 in this case. Each time a new color is written to a pixel, the global atomic counter is increased by one and a new node is written into the position indicated by the counter in the node buffer, with its tail pointer pointing to the position indicated by the head pointer of the pixel. If the head pointer has value -1, the tail pointer keeps its initial value -1. Then the head pointer is set to the position of the new node in the node buffer.

Spheres in Linked Lists When applied to order-independent transparency, a GPU linked list needs 3 elements (color, depth and tail pointer) per hit point between the ray and the surface of a rendered object. This results in 12 bytes per hit point if each element requires 32 bits. In our work, we need only 16 bytes *for each pair* of hit points (enter and leave of ray-sphere-intersection). We first render all spheres and insert their enter and leave depth values in linked lists. To save GPU memory, we only store the following information for each list entry: Enter and leave depth ($2 \times$ float), the sphere index ($1 \times$ int) and the tail pointer (using 32 bits for each). The sphere index allows the implementation of a selection function, such that the user can select one overlap per mouse click and get information about the spheres that form the overlap. Besides, the sphere index is used to read the center position ($3 \times$ float) and the basic color ($1 \times$ int) for one sphere. Thus the position, normal, and color of an enter/leaving point can be calculated based on the

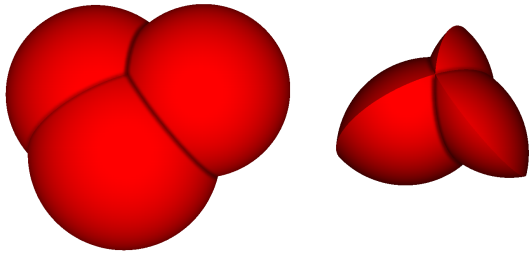

Figure 4: *Three spheres and the corresponding overlap rendering.*

depth value on the fly for later illumination. The depth values are stored as positive, linear z values in eye coordinates. For each of our visualizations, we first sort the depth values of each linked list along the z-axis until the surface for overlaps/gaps is found. Except of transparent renderings, we stop the sorting process at this point. We therefore need a sorting algorithm that copies the smallest element to front in each iteration, since this enables an early break of the sorting. In our work, Heapsort is applied here, since this is a non-recursive, in-place sorting algorithm which extracts the smallest element in $O(\log(n))$ steps. Since we allow both a container with hard and periodic boundary conditions, we have to differentiate between them for visualization.

5.1. Rendering Overlaps

To detect whether there are overlapping spheres along a camera ray, we make the following observation: In case of non-overlapping spheres, we have *alternating* enter and leave points along the camera ray. Whenever two spheres are overlapping, this pattern is changing and we observe *two successive enter points*. The second enter point is the one we need to draw, as can be seen in Figure 3. The whole process is summarized in algorithm 1. Figure 4 shows a small rendering with a few spheres and the resulting overlaps. The resulting "lenses" effectively describe a CSG intersection operation.

List 1: Inputs for algorithm 1, 2 and 3. All depth values are in the Eye coordinate system (ECS), assuming positive depth values along the z-axis. Tail pointers are first copied from global memory to register memory as T, such that swaps in the sorting algorithm are conducted on register memory and no writing-process to global memory is needed.

Data: T : array of tail pointers of the current pixel in register memory.
Data: L : length of T.
Data: D : array of vec2s. Each element stores the enter depth and the leave depth of the pixel ray and a sphere.
Data: $c.enter$: enter depth of the container.
Data: $c.leave$: leave depth of the container.

Algorithm 1: Draw overlaps for each pixel. Inputs are found in list 1.

Result: Determine the depth d of the frontmost overlap surface for the current pixel.

```
// at least two spheres are required for overlap
if L ≤ 1 then discard ;
// Initialization.
found ← false;
// y records the largest depth of leave points up to now
y ← 0;
// copy the tail pointer with the smallest enter depth to front
sortOneStep(0);
// loop through all spheres
for i ← 1 to L − 1 do
    // copy the tail pointer with the ith smallest enter depth to position i
    sortOneStep(i);
    // set d to the depth of the ith smallest enter point
    d ← D[T_i].enter;
    // set y to leave depth of previous sphere if larger
    y ← max{y, D[T_{i−1}].leave};
    // if previous leave depth is larger than current enter depth...
    if d < y then
        found ← true ;          // ...we found an overlap at depth d
        break;
if found then return d ;
else discard ;
```

5.2. Rendering Overlaps under Periodic Boundary Conditions

In case of periodic boundary conditions (PBCs), spheres can leave the container on each side and enter the container at the opposite side during collective rearrangement (wrap-around). PBCs have been applied extensively in theoretical modeling of crystalline solids [MP95], electrostatic systems [dLPS80] and biomolecular systems [CMF*95] etc., since a large (infinite) system can be approximated by using a small part called unit cell (container).

For a correct visualization, we need to make a *virtual copy* of each sphere which intersects the border to make sure that we display all overlaps with spheres from the opposite border. Since the periodic boundary results in overlapping spheres *outside* the container we have to extend our overlap test: Detected overlaps outside the container are either discarded or projected to the container border. Figure 5 shows an example with the different cases. The whole process is summarized in algorithm 2.

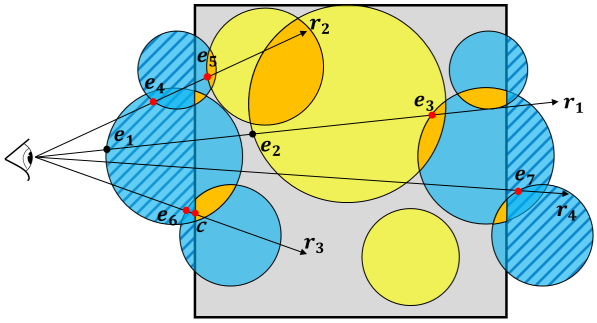

Figure 5: *Different cases for rendering overlaps under PBCs. Spheres intersecting the border are virtually duplicated and drawn in blue. The parts outside of the container are marked with stripes and emerge on the opposite border. Overlaps inside the container are drawn in orange. Ray r_1 shows an overlap inside the container. Detected overlaps outside the container can be discarded: Ray r_2 detects an overlap at e_4 which is skipped and e_5 is drawn. For ray r_4, the detected overlap at e_7 is ignored because it is behind the container. Ray r_3 detects at overlap in front of the container which is projected to the border ($e_6 \rightarrow c$).*

5.3. Rendering Gaps under Periodic Boundary Conditions

During collective rearrangement, it is interesting to see the empty regions where the overlapping spheres can still move into. Therefore we would like to *invert* the sphere rendering and visualize the surrounding empty space between the spheres. We use this type of visualization mainly for a periodic boundary, since a hard boundary effectively shows the container. In case of a periodic boundary, several spheres intersect the border and the visualization then shows the space between them. The basic idea to visualize this empty space is as follows: If the first intersection point along the camera rays is the container boundary, we are finished. If we first hit a sphere (which is outside the container due to the periodic boundary), we travel along the ray until we find a leave point which is *not in an intersection* between spheres. If this point is inside the container, we found the backside of the sphere that we are searching for. This point can then be rendered as the frontside of the empty space and illuminated with the inverted normal. Figure 6 shows a small example with different cases, the whole process is summarized in algorithm 3.

Figure 7 shows a small rendering with the empty space of two spheres. This type of rendering leads to a "cheese-like" appearance, where spheric holes are extracted from the container cube, similar to a CSG subtraction operation.

Algorithm 2: Draw overlaps for each pixel under PBCs. Inputs are found in list 1.

Result: Determine the depth d of the frontmost overlap surface for the current pixel.

```
// at least two spheres are required for overlap
if L ≤ 1 then discard ;
// initialization
found ← false;
inBox ← false;
// y records the largest depth of leave points up to now
y ← 0;
// copy the tail pointer with the smallest enter depth to front
sortOneStep(0);
// loop through all spheres
for i ← 1 to L − 1 do
    // copy the tail pointer with the ith smallest enter depth to position i
    sortOneStep(i);
    // set d to the depth of the ith smallest enter point
    d ← D[T_i].enter;
    // set y to leave depth of previous sphere if larger
    y ← max{y, D[T_{i−1}].leave};
    if d > c.enter then                    // if the enter point is in the container
        if d < y then                      // and overlap at d detected: stop
            found ← true;
            break;
    // if ray is in container after leaving current sphere
    else if D[T_i].leave > c.enter then
        // if current overlap projects to the container
        if y > c.enter and d < y then
            found ← true;                  // overlap found
            inBox ← true;                  // at front border
            break;
if inBox then return c.enter ;             // overlap at front border
else if d > c.leave then discard ;         // overlap behind container
else if found then return d ;              // overlap inside container
else discard ;                             // no overlap
```

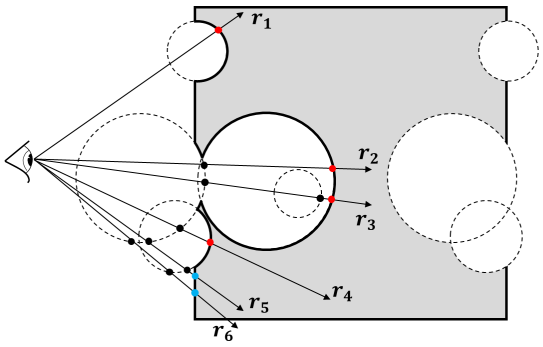

Figure 6: *Different cases for rendering gaps under PBCs. Black points are visited by corresponding rays and red points stand for the surface points of gaps. Ray r_1 directly detects the first leave point as the front surface of the empty space. For rays r_2 up to r_4, several leave points must be checked until the first leave point without sphere intersection is found. Rays r_5 and r_6 detect leave points in front of the container and are therefore discarded.*

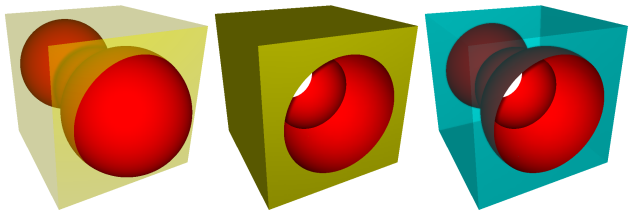

Figure 7: *Rendering gaps for two spheres under PBCs: One sphere intersects the container border (left), so it is duplicated to the opposite border. Center: Resulting empty space rendering. Right: Empty space rendering with transparent container.*

5.4. Discussion

For both the overlaps and gaps visualization, we generate a geometry buffer, illuminate and apply screen-space ambient occlusion (similar to direct sphere rendering). Both visualizations are also applicable for any convex primitives whose enter and leave points can be calculated. Furthermore, our methods can also be extended to render overlaps/gaps or to calculate their volumes for arbitrary 2-manifold 3D-meshes. Either with two render passes (one for front side, one for back side) or one render pass (with depth test disabled). When one render pass is applied, we need four elements (depth, index, normal, tail pointer) per hit point and sort all hit points. Besides, to render gaps or to calculate volumes of gaps/overlaps, an extra array in register memory is needed to record the indices of points visited by the camera ray so that leave points can be recognized.

6. Volume Calculation

During collective rearrangement we would like to judge whether the quality of the simulation still improves. In this case, overlapping spheres are still moving into the remaining free spaces. This means that both the total amount of overlap and the amount of remaining free space decrease. The required analytic computation of the intersection volume of multiple spheres is possible, but complicated. Instead, we use the linked lists per pixel to compute the total free volume and the total overlap volume at pixel precision. The basic idea is to compute the volume for each pixel as a sum of cuboid (orthographic) or frustum (perspective) volumes, resulting from the stored depth values. If the used sphere packing contains spheres smaller than a pixel, we use *tiled rendering* in combination with an orthographic projection. The tile resolution is then adjusted, such that the smallest spheres are still larger than a pixel.

In the following, we assume a viewport resolution of $w \times h$ pixels. To determine the volume V of a pixel, we start with 0 and compute the volume by iterating through the linked list. The total volume is then determined as the sum of all pixel volumes.

Algorithm 3: Draw gaps for each pixel under PBCs. Inputs are found in list 1.

```
Result: Determine the depth d of the free space surface for the
        current pixel under PBCs.
// discard in case of no spheres (container has been rendered already)
if L = 0 then discard ;
// initialization
inBox ← false;
// copy the tail pointer with the smallest enter depth to front
sortOneStep(0);
d ← D[T_0].leave;          // set d to the first possible value for a gap
if d > c.enter then inBox = true ;  // check if d is in the container
// loop through all spheres
for i ← 1 to L − 1 do
    // copy the tail pointer with the ith smallest enter depth to position i
    sortOneStep(i);
    if inBox then
        // if sphere i enter depth is behind prev. sphere leave depth:
        // gap found at backside of prev. sphere
        if D[T_i].enter > d then break;
        // set d to leave depth of current sphere i if larger
        else d ← max{d, D[T_i].leave} ;
    else
        // if leave depth of prev. sphere is in container
        if d > c.enter then
            // if sphere i enter depth behind prev. sphere leave depth
            // gap found at backside of prev. sphere
            if D[T_i].enter > d then break;
            // set d to leave depth of current sphere i if larger
            else d ← max{d, D[T_i].leave} ;
            inBox = true;
        // if gap between sphere i and prev. sphere before container
        else if D[T_i].enter > d then
            // if enter depth of sphere i is in container: draw border
            if D[T_i].enter > c.enter then discard;
            // else continue with leave depth of sphere i
            else d ← D[T_i].leave;
        // set d to leave depth of current sphere i if larger
        else d ← max{d, D[T_i].leave} ;
if d > c.leave then discard ;   // ignore gap behind the container
else return d ;
```

6.1. Volume Calculation under Orthographic Projection

If orthographic projection is applied, the area A_p of each pixel is simply

$$A_p = \frac{(t-b)(r-l)}{hw} \tag{1}$$

where t, b are coordinates for the top and bottom horizontal clipping planes and r, l are coordinates for the right and left vertical clipping planes.

Therefore, for each pair of matched enter point with depth d_e and leave point with depth d_l, the difference $d_l - d_e$ is added to V and the sum of V over all pixels is finally multiplied by A_p given by equation (1) to calculate the approximated volume.

6.2. Volume Calculation under Perspective Projection

Using perspective projection with field of view angle θ and viewport aspect ratio α, the area A_p of each pixel corresponds to a rectangle with side length $2z\tan(\theta/2)/h$ in y direction and the side length $\alpha 2z\tan(\theta/2)/w$ in x direction. The pixel area can therefore be calculated as

$$A_p = \frac{4z^2\alpha\tan(\theta/2)^2}{hw} \tag{2}$$

where z is the depth value in the eye coordinate system of the point belonging to the current pixel.

Under perspective projection, for each pair of matched enter point with depth d_e and leave point with depth d_l, the difference $d_l^3 - d_e^3$ is added to V and the sum of V over all pixels is finally multiplied by $4\alpha\tan(\theta/2)^2/(hw)$ based on equation (2) to calculate the approximated volume.

7. Results

All performance measurements were conducted on an NVIDIA GeForce GTX 1080 graphics card with 8 GB video RAM. The PC is equipped with an Intel Core I7-6700K processor, 4.00 GHz CPU, 32 GB RAM, running Windows 10 (x64). If not indicated otherwise, the viewport resolution is set to 768×768 pixels. Please see also the accompanying video for real-time frame captures.

Figure 8 shows our renderings of gaps and overlaps for different numbers of spheres. For rendering the whole container with one million spheres, the frame rate is 58 fps for the standard rendering and 45 fps for our overlaps and gaps visualization. In combination with the CR simulation, the frame rate drops to 20 fps for standard rendering and 18 fps for overlaps and gaps. Including the volume calculation requires approximately 17 ms additional time. The memory requirements for storing the linked lists are approx. 161 MB. Our largest data set contains ten million spheres. Here, we still reach a frame rate of 17 fps for the standard rendering and 11 fps for gaps and overlaps visualization. We observed that both the rendering time and the memory requirements increase linearly with the viewport resolution. Furthermore, the memory requirements increase linearly with the summed cross sectional area of all spheres. A detailed analysis of both timings and memory requirements is given in the supplemental material.

In certain situations, it can be helpful to see a combination of overlaps and gaps in the same image. Figure 9 shows such a rendering (47 fps for one million spheres). Here, gaps from large spheres are visible as well as their overlaps with many, small surrounding spheres "on the border" of the gaps.

The usual way to see gaps and overlaps between objects is the insertion of 2D clipping planes. Figure 10 shows a comparison between a 2D clipping plane and our 3D visualization of overlaps. Note that it is difficult to get an overview of the distribution and shape of the overlaps only based on a moving 2D clipping plane. When combined with a clipping plane, our method also shows the overlaps on the clipping plane, but in addition, the 3D structure of the overlaps behind the clipping plane. We found this a more useful combination than flying inside the 3D structure, which can become confusing. Please see also the accompanying video to spot

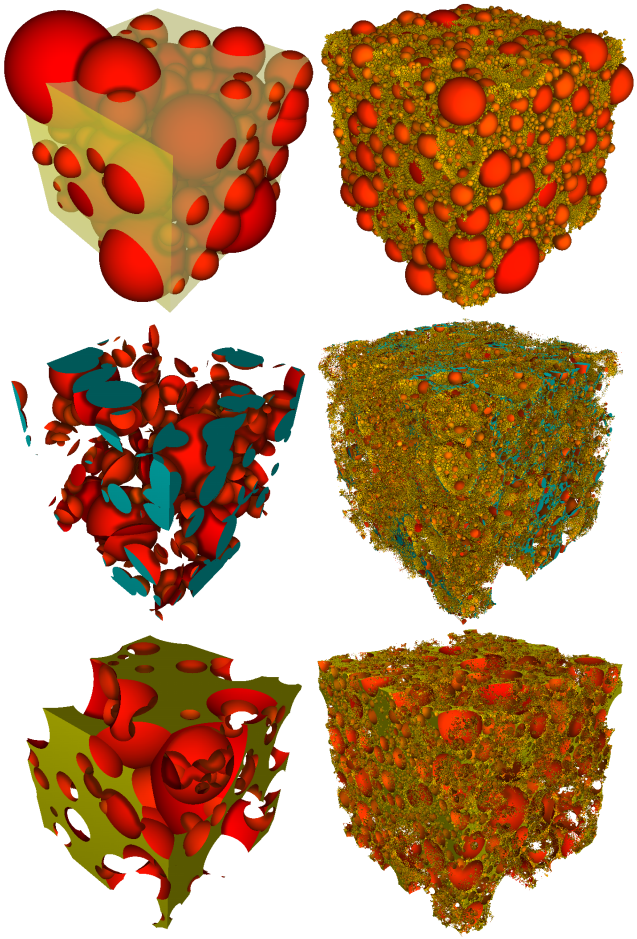

Figure 8: *Overlaps (center row) and gaps (bottom row) for 128 spheres (left) and one million spheres (right). Please note that PBCs are used here, therefore some gaps and overlaps can result from spheres at the opposite border. The transparent yellow cube is the container, overlaps on the container border are drawn in blue.*

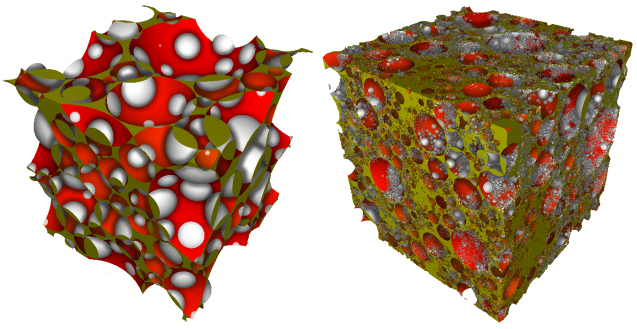

Figure 9: *Combined rendering with both overlaps (drawn in grey) and gaps (drawn in red) for 128 spheres (left) and one million spheres (right).*

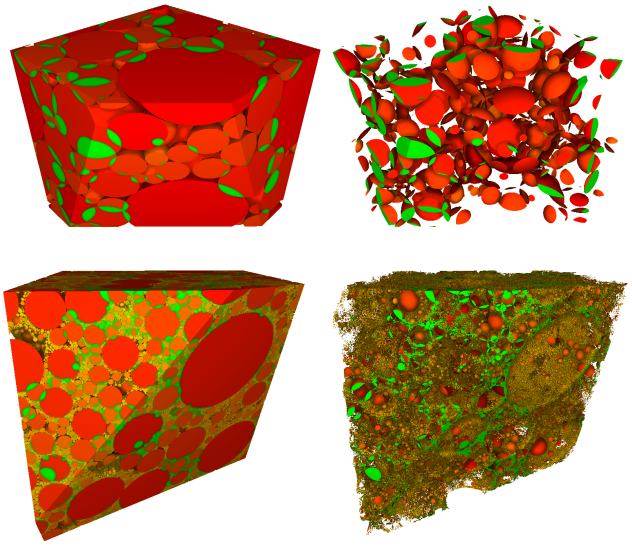

Figure 10: *Comparison of our 3D overlap rendering with standard 2D clipping planes for 128 spheres (top) and one million spheres (bottom). Overlaps at the clipping plane and at the container border are drawn in green.*

the difference between the different approaches and also the combination of our gap and overlap visualizations with a clipping plane. In addition, the video shows renderings with transparency and a user-defined virtual "stirring" to resolve overlaps that remain after the CR simulation.

8. Conclusion and Future Work

We presented a new visualization technique to display overlaps and gaps in large sphere packings. For collective rearrangement simulations, this allows the user to inspect the quality of the simulation in each iteration. Our method reaches interactive to real-time frame rates for millions of spheres. Showing these structures in 3D gives a better impression of their shape than using 2D clipping planes.

As future work, we intend to integrate more realistic particle shapes, like polyhedrons [GRZ*10] or collections of spheres, both in the visualization and the simulation. In case of even larger sphere packings, with many spheres that are smaller than a pixel, we examine if ray casting in combination with the loose octree can be used to display the gaps and overlaps more quickly [FKE13]. We investigate if additional visualization techniques, like depth-aware contours or depth darkening can improve the perception of the gaps and overlaps rendering [MTC06]. Since we detect the position of overlaps and gaps regions, repelling forces could be added in the overlap regions, as well as attracting forces in the gaps regions, to automatically improve the simulation, which could lead to a higher density of the sphere packing. Furthermore, we plan to use the visualization for other types of simulations, like random sequential addition, where spheres are attracted by gravity and roll down until they have at least three contact points, which often leaves large gaps between the spheres.

9. Acknowledgments

Feng Gu receives a scholarship of the Simulation Science Center Clausthal- Göttingen within the project 'Virtual Microscope'. Zhixing Yang is supported by Dyckerhoff Stiftung, grant number T218/26441/2015.

References

[BBS*02] Bezrukov A., Bargiel M., Stoyan D., et al.: Statistical analysis of simulated random packings of spheres. *Particle & Particle Systems Characterization 19*, 2 (2002), 111. 1

[CMF*95] Cheatham T. I., Miller J., Fox T., Darden T., Kollman P.: Molecular dynamics simulations on solvated biomolecular systems: the particle mesh ewald method leads to stable trajectories of dna, rna, and proteins. *Journal of the American Chemical Society 117*, 14 (1995), 4193–4194. 4

[dLPS80] de Leeuw S. W., Perram J. W., Smith E. R.: Simulation of electrostatic systems in periodic boundary conditions. i. lattice sums and dielectric constants. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* (1980), vol. 373, The Royal Society, pp. 27–56. 4

[FKE13] Falk M., Krone M., Ertl T.: Atomistic visualization of mesoscopic whole-cell simulations using ray-casted instancing. *Computer Graphics Forum 32*, 8 (2013), 195–206. URL: http://dx.doi.org/10.1111/cgf.12197, doi:10.1111/cgf.12197. 2, 7

[GKM*15] Grottel S., Krone M., Müller C., Reina G., Ertl T.: Megamol – a prototyping framework for particle-based visualization. *Visualization and Computer Graphics, IEEE Transactions on 21*, 2 (2 2015), 201–214. doi:10.1109/TVCG.2014.2350479. 2

[GRZ*10] Grottel S., Reina G., Zauner T., Hilfer R., Ertl T.: Particle-based rendering for porous media. In *Proceedings of SIGRAD 2010: Content aggregation and visualization; November 25–26; 2010; Västerås; Sweden* (2010), no. 052, Linköping University Electronic Press, pp. 45–51. 7

[KD04] Kirsch F., Döllner J.: Rendering techniques for hardware-accelerated image-based CSG. In *The 12-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2004, WSCG 2004, University of West Bohemia, Campus Bory, Plzen-Bory, Czech Republic, February 2-6, 2004* (2004), pp. 221–228. 2

[KKF*16] Kozlikova B., Krone M., Falk M., Lindow N., Baaden M., Baum D., Viola I., Parulek J., Hege H.-C.: Visualization of biomolecular structures: State of the art revisited. *Computer Graphics Forum* (2016), n/a–n/a. URL: http://dx.doi.org/10.1111/cgf.13072, doi:10.1111/cgf.13072. 2

[KRW10] Kolonko M., Raschdorf S., Wäsch D.: A hierarchical approach to simulate the packing density of particle mixtures on a computer. *Granular Matter 12*, 6 (2010), 629–643. URL: http://dx.doi.org/10.1007/s10035-010-0216-5, doi:10.1007/s10035-010-0216-5. 1, 2

[MP95] Makov G., Payne M.: Periodic boundary conditions in ab initio calculations. *Physical Review B 51*, 7 (1995), 4014. 4

[MPSV14] Muzic M. L., Parulek J., Stavrum A.-K., Viola I.: Illustrative visualization of molecular reactions using omniscient intelligence and passive agents. *Computer Graphics Forum 33*, 3 (June 2014), 141–150. Article first published online: 12 JUL 2014. URL: https://www.cg.tuwien.ac.at/research/publications/2014/lemuzic-2014-ivm/. 2

[MTC06] Montani C., Tarini M., Cignoni P.: Ambient occlusion and edge cueing for enhancing real time molecular visualization. *IEEE Transactions on Visualization and Computer Graphics 12* (2006), 1237–1244. doi:doi.ieeecomputersociety.org/10.1109/TVCG.2006.115. 2, 7

[RFV13] Rossignac J., Fudos I., Vasilakis A.: Direct rendering of boolean combinations of self-trimmed surfaces. *Computer-Aided Design 45*, 2 (2013), 288–300. URL: https://doi.org/10.1016/j.cad.2012.10.012, doi:10.1016/j.cad.2012.10.012. 2

[RGS09] Ritschel T., Grosch T., Seidel H.-P.: Approximating Dynamic Global Illumination in Image Space. In *Proceedings ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D) 2009* (2009), pp. 75–82. 2

[RK09] Raschdorf S., Kolonko M.: *Loose octree: a data structure for the simulation of polydisperse particle packings*. Tech. rep., TU Clausthal, 2009. 2

[RK11] Raschdorf S., Kolonko M.: A comparison of data structures for the simulation of polydisperse particle packings. *International journal for numerical methods in engineering 85*, 5 (2011), 625–639. 2

[RN14] Raza J. L. G., Nunes G.: Screen-space deformable meshes via csg with per-pixel linked lists. In *GPU Pro 5*, Engel W., (Ed.). CRC Press, 2014, pp. 233–240. 2

[Tor06] Torquato S.: *Random Heterogenous Materials: Microstructure and Macroscopic Properties*, 2nd ed., vol. 16 of *Interdisciplinary Applied Mathematics*. Springer, 2006. 1

[YGGK17] Yang Z., Gu F., Grosch T., Kolonko M.: *Accelerated Simulation of Particle Packings Using Parallel Hardware*. Tech. rep., TU Clausthal, 2017. 1, 2

[YHGT10] Yang J. C., Hensley J., Grün H., Thibieroz N.: Real-time concurrent linked list construction on the gpu. In *Proceedings of the 21st Eurographics Conference on Rendering* (Aire-la-Ville, Switzerland, Switzerland, 2010), EGSR'10, Eurographics Association, pp. 1297–1304. URL: http://dx.doi.org/10.1111/j.1467-8659.2010.01725.x, doi:10.1111/j.1467-8659.2010.01725.x. 2, 3