Eurographics Workshop on Visual Computing for Biology and Medicine (2019)
B. Kozlíková, L. Linsen, and P.-P. Vázquez (Editors)

Interactive CPU-based Ray Tracing of Solvent Excluded Surfaces

T. Rau[1†], S. Zahn[3], M. Krone[4], G. Reina[1], and T. Ertl[2]

[1]University of Stuttgart, VISUS, Germany
[2] University of Stuttgart, VIS/VISUS, Germany
[3] University of Stuttgart, Germany
[4] University of Tübingen, Big Data Visual Analytics, Germany

Abstract

Depictions of molecular surfaces such as the Solvent Excluded Surface (SES) can provide crucial insight into functional molecular properties, such as the molecule's potential to react. The interactive visualization of single and multiple molecule surfaces is essential for the data analysis by domain experts. Nowadays, the SES can be rendered at high frame rates using shader-based ray casting on the GPU. However, rendering large molecules or larger molecule complexes requires large amounts of memory that has the potential to exceed the memory limitations of current hardware. Here we show that rendering using CPU ray tracing also reaches interactive frame rates without hard limitations to memory. In our results large molecule complexes can be rendered with only the precomputation of each individual SES, and no further involved representation or transformation. Additionally, we provide advanced visualization techniques like ambient occlusion opacity mapping (AOOM) to enhance the comprehensibility of the molecular structure. CPU ray tracing not only provides very high image quality and global illumination, which is beneficial for the perception of spatial structures, it also opens up the possibility to visualize larger data sets and to render on any HPC cluster. Our results demonstrate that simple instancing of geometry keeps the memory consumption for rendering large molecule complexes low, so the examination of much larger data is also possible.
(see https://www.acm.org/publications/class-2012)

CCS Concepts

• ***Human-centered computing*** → ***Scientific visualization;*** • ***Computing methodologies*** → ***Ray tracing;*** • ***Applied computing*** → ***Molecular structural biology;***

1. Introduction

Researchers studying biomolecules are interested in the potential of interaction between different molecules. These biomolecules have certain regions of interest with special characteristics, for example the binding sites of a protein, where a specific small molecule can trigger an enzymatic reaction. Another example is docking, where two molecules that fit together like two pieces of a puzzle form a larger molecular complex. Molecular dynamics simulation is an important tool to study such molecular interactions. A visualization that allows domain experts to explore the simulation results supports them in drawing conclusions about the simulated molecules. Therefore, a multitude of molecular models have been defined, each one highlighting a specific aspect of a molecule, for example, the molecular volume, covalent bonds between atoms, or the molecular surface. Among the different molecular surfaces, the *solvent excluded surface* (SES) is of high importance as it highlights exposed areas of biomolecules that can be reached by other surrounding molecules. It provides an intuitive representation of the interface between a molecule and its surroundings.

The generation and rendering of the SES is not trivial and involves advanced algorithms and special cases. Several approaches exist to compute and visualize the SES [KKF*17]. We base our computation on the *contour-buildup* (CB) algorithm [TA96], which can be efficiently parallelized [LBPH10, KGE11]. The resulting surface is an algebraic representation that consists of three different geometric primitives. In order to attain high image quality and semi-transparent rendering of the SES for the visual analysis, we use ray tracing instead of rasterization, which would be the common choice for interactive visualization. Intersections between the geometric primitives of the SES and a viewing ray can be obtained by just finding the root of one equation per primitive. Semi-transparent rendering is especially useful for the visual analysis, as it allows domain experts to simultaneously see the molecular surface and see what is underneath it. Technically, this involves simple blending and requires that the rays are not terminated after the first intersection is found. However, semi-transparent rendering of the SES requires a clear definition of the geometric primitives' visible parts, which is not straightforward. Ray tracing can also be used for

† tobias.rau@visus.uni-stuttgart.de

DOI: 10.2312/vcbm.20191249

feature extraction. One example is to highlight cavities of the SES using *ambient occlusion opacity mapping* (AOOM) [Bor11].

Modern interactive visualization often relies on the parallel computing power offered by the GPU. However, running visualizations on large data can quickly reach the limit of todays GPU memory. Furthermore, high performance computing (HPC) systems are often not equipped with dedicated GPUs. Therefore we use a CPU rendering option to achieve interactive frame rates and high-quality images. The CPU ray tracing engine OSPRay [WJA*17] offers all required functionality for this task. Additionally, efficient spatial acceleration structures have the potential to reach higher frame rates than rasterization, especially for large data sets. Ray tracing simplifies rendering of realistic illumination effects, while still offering interactivity [WMG*09].

The ray tracing technique used for rendering comes with another advantage: obtaining high quality renderings of the SES with, for example, global illumination of the scene does not require additional effort. The global illumination and resulting occlusion increases the depth perception and structural perception of renderings — an important requirement when rendering molecular surfaces. The spacial structure is easily intelligible as *ambient occlusion* (AO) is darkening cavities of the molecules [Bor11, TCM06].

Our contributions presented in this paper can be summarized as follows: We present a CPU alternative to interactive SES rendering that is capable of rendering single molecules and molecule complexes. To reach this goal, we implemented our approach using the OSPRay framework. Furthermore, we used the ray tracing to specifically highlight surface features such as cavities, which are important for molecular function. Our solution is easily deployed on any HPC system for e. g., in-situ coupling to a simulation to achieve higher frame rates using distributed rendering.

2. Fundamentals & Related Work

In this section, we briefly introduce molecular surface visualization and explain algorithms we base our work on. We then detail the optimizations we introduce to the various processing stages required for rendering. We also discuss the mathematical background and the software packages we use for the work we present in this paper.

2.1. Molecular Surfaces

As mentioned above, molecular surfaces are an important representation for the analysis of molecular interactions. Kozlíková et al. [KKF*17] recently presented an in-depth survey of the different types of molecular surfaces, their computation, and their visualization. Therefore, we only briefly introduce the surface types that are necessary for the definition of the SES, which we used in this work. This surface is commonly used for the visual analysis of biomolecules.

The *van der Waals surface* (vdW) is the most simple molecular surface. It represents each atom by a sphere. The radius of each sphere equals the van der Waals radius of the corresponding atom [KKF*17], which can be derived from the interaction parameters of the *Lennard-Jones* potential.

Another simple representation is the *solvent accessible surface* (SAS) [Ric77]. This surface is constructed by rolling a probe sphere over the van der Waals surface; all possible center positions of this probe sphere are considered as the SAS. The radius of this probe is set to approximate a specific solvent (e.g., $R_{\text{probe}} = 1.4$ Å, a spherical simplification of H_2O). A simpler approach is to extend the radii of the atoms by the probe radius R_{probe}.

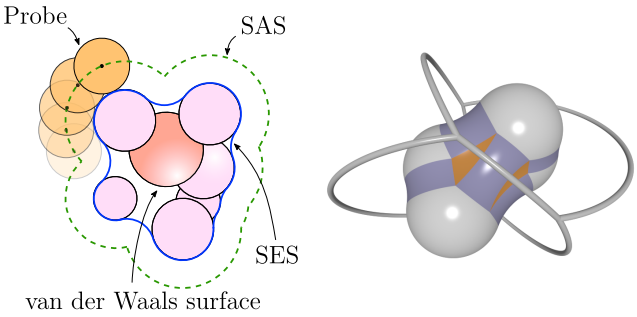

Figure 1: *On the left a sketch of the most common molecular surface types is depicted. Red and pink spheres are the atoms of a molecule with their corresponding vdW radius, orange spheres depict the rolling probe. The surface of the atoms directly defines the vdW surface. The path of the probe center (i.e., the Solvent Accessible Surface) is shown as a green dashed line. The blue line depicts the SES. On the right side an example rendering of the SES of a small molecule is shown. Convex spherical patches are rendered in light gray, toroidal patches in purple, and spherical triangles are depicted in orange. Additionally, the contours of the contour-buildup algorithm are shown as arcs around the molecule.*

The *solvent excluded surface* (SES) is constructed in a similar way as the SAS. Again a probe sphere is rolled over the van der Waals surface; however, the SES is defined by the surface patches the probe surface traces while rolling over the surface. The effect can be compared to shrink-wrapping the van der Waals surface. The surface is classified into two parts, the *contact* surface and the *reentrant* surface [Ric77]. The contact surface consists of all atom surfaces that the probe can touch while rolling over the vdW surface. Thus, all of the SES that is not a direct part of the vdW surface is the reentrant surface (the probe filling the gaps when it can get no further). While vdW surface and the SAS allow discontinuities in regions where spheres are in contact, the SES generates smooth connections between primitives. However, not all contact points provide a unique tangent plane, there are a few exceptions that result in singularities [Con83]. In Figure 1, the vdW surface, the SES, and the SAS are depicted.

The SES can be described analytically using three basic geometric primitives [Con83]:

Convex spherical patches are generated when the probe is on contact with only one sphere and can move freely on its surface (two degrees of freedom).

Toroidal patches are resulting from the probe touching two spheres simultaneously. The probe can only move on a circular arc (one degree of freedom).

Concave spherical patches/spherical triangles form when the

probe is in contact with three spheres. Here, the probe has no freedom to move (zero degrees of freedom).

Toroidal patches, for example, are always connected to convex spherical patches, hence the free 2D movement of the probe loses a degree of freedom when hitting a neighboring sphere. Also, three neighboring toroidal patches form a spherical triangle, as they are always found on connection points of the contour arcs that are generated by the contour-buildup algorithm. If the probe intersects the axis of rotation of an arc or an arc piece, a so-called spindle torus forms. The minor radius of this toroidal patch, is larger than the major radius and any connected spherical triangles will be singular as they intersect each other. In Figure 1 the different types of patches are depicted for a SES of a small molecule.

The SES can be computed in two ways: the first one is to extract the aforementioned geometric primitives and render them, while the second one is to discretize the volume of the molecule on a grid and extract it as an isosurface. The latter approach was for example used by Can et al. [CCW06] and Yu et al. [Zey09]. Recently, Hermosilla et al. [HKG*17] presented a progressive grid-based computation of the SES that runs on the GPU. Many solutions have been proposed for the analytic approach, starting with the work of Connolly [Con83], who devised the formulas that describe the geometric primitives. For a detailed overview of the different approaches, we refer to the survey of Kozlíková et al. [KKF*17]. We based our computation of the SES on the contour-buildup algorithm by Totrov and Abagyan [TA96], which was also used by Lindow et al. [LBPH10] and Krone et al. [KGE11], since it can be parallelized efficiently on multi-core CPUs and GPUs. A description of the algorithm and our optimizations is given below in Section 2.2. Another approach was introduced by Parulek and Viola [PV12] where the SES is rendered without any precomputation via implicit functions.

2.2. The Contour-Buildup Algorithm

The contour-buildup algorithm was introduced by Totrov and Abagyan [TA96] and computes the paths that the probe sphere can roll along with at most one degree of freedom. This algorithm is a per-sphere approach and therefore is easily parallelizable [KGE11, LBPH10], as mentioned above. On modern, parallel computing hardware, it is therefore preferable to inherently sequential methods like the *reduced surface* (RS) algorithm introduced by Sanner et al. [SOS96]. In the contour-buildup algorithm, the path that the probe center is following is either an arc or a full circle. The algorithm can be separated into two phases. In phase one full circles of intersecting spheres with the extended radius R_i' are calculated and in phase two these circles are split into the corresponding arcs on crossings. The extended radius of each atom R_i of the molecule is defined as

$$R_i' = R_i + R_{\text{probe}} \quad , \tag{1}$$

which corresponds to the SAS. Considering sphere σ_i has a set of neighbors $N(\sigma_i)$ that is in intersection range. The intersection circle c_j of spheres σ_i and $\sigma_j \in N(\sigma_i)$ is calculated via the sphere position

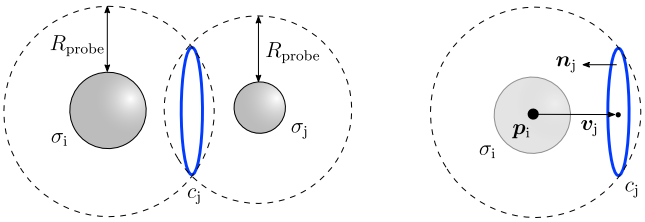

Figure 2: *Intersection of two probe radius extended spheres (dashed lines)* σ_i *and* σ_j*. The blue intersection circle is defined by the circle center* $\boldsymbol{v}_j$ *that is relative to the sphere position* $\boldsymbol{p}_i$*. The normal of the circle is denoted as* $\boldsymbol{n}_j$*.*

$\boldsymbol{p}_i$ and the circle center relative to the sphere position [TA96]

$$\boldsymbol{v}_j = \boldsymbol{v}_{ij} \frac{R_i'^2 + \boldsymbol{v}_{ij}^2 - R_j'^2}{2\,\boldsymbol{v}_{ij}^2} \quad , \tag{2}$$

where $\boldsymbol{v}_{ij}$ is defined as $\boldsymbol{p}_j - \boldsymbol{p}_i$. The normal to the circle results as $\boldsymbol{n}_j = -\boldsymbol{v}_{ij}$ and the intersection circle radius yields

$$r(c_j) = \sqrt{R_j' - \boldsymbol{v}_j^2} \quad . \tag{3}$$

Figure 2 shows a sketch of two intersecting spheres and the resulting parameters. If $N(\sigma_i)$ contains more than one element, the resulting intersection circle c_j has to be checked against all other intersection circles with sphere σ_i. Considering another circle c_k intersects with c_j the planes defined by $\boldsymbol{n}_j$ and $\boldsymbol{n}_k$ are also intersecting. The resulting intersection line is also intersecting both circles c_k and c_j. The calculation of the two intersection points yields to

$$\boldsymbol{x}_{1,2} = \boldsymbol{h} \pm \boldsymbol{a} \frac{R_i' - \boldsymbol{h}^2}{\boldsymbol{a}^2} \quad , \tag{4}$$

where $\boldsymbol{a} = \boldsymbol{v}_i \times \boldsymbol{v}_j$ is the normal of the two relative circle centers and $\boldsymbol{h}$ is the auxiliary position vector that is located in the middle between the two intersection points $\boldsymbol{x}_{1,2}$. However, in many cases the circles c_k and c_j are not intersecting each other. Therefore, $\boldsymbol{h}$ is used as an indicator if the circles are intersecting. If $\boldsymbol{h}$ is located inside the extended sphere of σ_i, the two circles are intersecting.

Considering circle c_k is not intersecting c_i, there are four different cases that need to be distinguished to be able to construct the SES correctly:

1. Sphere σ_k and sphere σ_j do not cover c_j and c_k respectively
2. Sphere σ_k covers circle c_j completely
3. Sphere σ_j covers circle c_k completely
4. Sphere σ_j covers circle c_k completely and sphere σ_k covers circle c_j completely

To reliably distinguish the cases showed above, Totrov and Abagyan [TA96] defined three quantities

$$g_1 = \boldsymbol{n}_j \boldsymbol{n}_k \quad , \tag{5}$$

$$g_2 = \boldsymbol{m}_j \boldsymbol{m}_k \quad , \tag{6}$$

$$g_3 = \boldsymbol{n}_j \boldsymbol{q} \quad , \tag{7}$$

with $\boldsymbol{m}_i = \boldsymbol{v}_i - \boldsymbol{h}$ and $\boldsymbol{q} = \boldsymbol{v}_k - \boldsymbol{v}_j$. Depending on the sign of the parameters g_1, g_2, and g_3, the previously defined cases appear (see Table 1).

Table 1: *All combinations of the three quantities that describe the configuration of two circles.*

$g_1 > 0$	$g_2 > 0$	$g_3 > 0$	Case
true	true	true	2
true	true	false	3
true	false	true	1
true	false	false	4
false	true	true	1
false	true	false	4
false	false	true	2
false	false	false	3

In the first case two circles c_j and c_k are generated, however those circles can also cross each other. The crossing points are calculated and the circles are split into arcs. In cases two and three a circle is completely covered by a sphere and therefore these circles are deleted. The sphere σ_i is completely inside sphere σ_j and σ_k and is removed form the further contour calculation.

This also shows the drawback of the contour-buildup algorithm. The sorting showed in Table 1 transitions into excessive branching in the implementation. Therefore, a *SIMDization* of the contour-buildup algorithm does not increase its performance.

2.3. Generating the Surface

Sphere primitives are generated if the corresponding sphere σ_i is not covered. Toroidal patches are generated from the arcs of the contour with a minor radius of R_{probe} and a major radius of c_j (see Figure 2). Also the center of the toroidal patch is defined by the center of circle c_j. To ensure that this geometry is not generated twice, a patch is only created when the index of the current sphere is smaller than the index of the sphere that is touched by the probe sphere. A spherical triangle is generated at points where three arc endpoints meet. The radius of this patch is then R_{probe} and a similar index comparison as for the toroidal patches is used to prevent double or in this case triple generation of the same geometry. It is possible that a spherical triangle can connect to a toroidal patch with a minor radius larger than the major radius. In this case, parts of the adjacent spherical triangles are overlapping. This is handled by cutting spherical triangles with probe spheres of surrounding spherical triangles.

2.4. Rendering of Algebraic Surfaces

An efficient method to render the resulting algebraic geometries is ray tracing. Rays are generated from an eye position and shot through an image plane into the scene. The intersection points between these view rays and the geometry determine the visibility of the objects in the scene. For algebraic surfaces of low polynomial order, these intersections can be computed analytically in a fast and precise manner. Modern GPU-based ray casting [Gum03, RE05] combines this idea with traditional rasterization: only a simple bounding geometry is rendered for each object and the actual ray-object intersections are computed per pixel in the fragment shader. Note that this method only uses primary rays that originate from the eye; secondary rays that would be needed for shadows and other global illumination effects are not efficient, since the standard approach does not maintain a queryable global data structure. There are variants, like the approaches by Lindow et al. [LBH12] or the approach by Falk et al. [FKE13] that are grid-based, such that secondary rays can be efficient. As mentioned above, implicit surfaces have been found to be more efficient than traditional triangle-based rendering on modern GPUs. The technique is, therefore, widely used in molecular visualization of models that consist of low-order polynomial surfaces like spheres, cylinders, or tori [KKF*17].

However, high image quality can only be achieved if the ray is not simply terminated after the first hit. Global illumination effects such as shadows or ambient occlusion lead to a realistic rendering of the scene. Hence, the computational effort of ray tracing lies in the ray-geometry intersection tests. For acceleration, the geometries in the scene are segmented into hierarchical structures such as bounding volume hierarchies (BVH). However, a hierarchy has to be built before the scene can be rendered, and thus incapacitates time dependent data. Several frameworks have been created for scientific visualization to efficiently calculate the ray-geometry intersection.

Wald et al. [WJA*17] present their ray tracing engine OSPRay that is optimized for running on Intel CPUs. Stone et al. [SMSS16] showcased a visualization framework that is able to run in-situ with a *molecular dynamics* (MD) simulation. However, this approach completely relies on graphics acceleration hardware. Parker et al. [PBD*10] developed the OptiX framework for efficient ray tracing on Nvidia GPUs. Both interactive ray tracing approaches are used for molecular visualization, for example by the popular molecular visualization tool VMD [HDS96]. In contrast to our solution, the ray tracing of the SES offered by VMD does not use the algebraic description of the surface patches, but rather uses a triangle mesh computed by MSMS [SOS96]. Another completely different approach is presented by Bruckner [Bru19]. The author defines a *Gaussian molecular surface* that is derived from a density function of the atom positions. Using visibility information and on-the-fly sorting, this technique is able to interactively render dynamic molecular data.

Krone et al. [KBE09] where the first to use GPU-based ray casting to render the patches of the analytically described SES. This rendering technique, which was introduced by Gumhold [Gum03], is faster and more efficient than traditional tessellation of the patches into triangle meshes. However, it makes rendering the SES semi-transparently even more challenging, since the interior parts of the surface geometries (torus patches and spheres) have to be cut away correctly. Kauker et al. [KKP*13] presented a method that uses constructive solid geometry; however, their algorithm used the Reduced Surface [SOS96], which is not parallelizable. Recently, Jurcik et al. [JPSK16] extended the work of Krone et al. [KGE11] that uses an efficient GPU-parallelization of the contour-buildup algorithm, which allowed to render the SES semi-transparently. In Section 3, we explain our semi-transparent SES rendering. We decided to base our implementation on OSPRay for its independence from GPUs.

2.5. The OSPRay Ray Tracing Engine

OSPRay is an open source CPU ray tracing engine and builds upon the kernels of Embree [WWB*14]. It provides abstract structures such as renderers, materials, lights, and geometries. Embree efficiently builds and traverses acceleration structures (axis aligned BVH). Depending on the architecture, the traversal is either vectorized over the BVH nodes or the components of the intersection variables. OSPRay's modular structure also allows researchers to implement their own extensions. Custom geometries have to provide call back functions for intersection tests and bounding box queries. Currently, these functions have to be implemented using the Intel SIMD compiler (ISPC) [PM12] to accelerate the performance critical code of OSPRay. GPUs are still rare on current HPC systems, therefore OSPRay is a viable alternative to classical GPU rasterization. Additionally, by using ray tracing, the implementation of global effects is less complicated than for rasterization. In our work, we use OSPRay as built into the visualization framework MegaMol [RKRE17, GKM*15].

2.6. Semi-Transparent SES Rendering

For primitive-wise GPU ray casting, as is commonly used for transparent renderings of the SES [KKL*16], objects must be rendered in order - either from back to front or vice versa - for correct blending results. Starting with the most distant fragment for the blending process (back to front) is described by Bavoil et al. [BM08]

$$C_{\text{dst}} \leftarrow A_{\text{src}} C_{\text{src}} + (1 - A_{\text{src}}) C_{\text{dst}} \quad , \tag{8}$$

where C_{src} and C_{dst} are the source and destination colors, and A_{src} is the source opacity. A different result is obtained by using front to back rendering

$$C_{\text{dst}} \leftarrow A_{\text{dst}} (A_{\text{src}} C_{\text{src}}) + C_{\text{dst}} \quad , \tag{9}$$

with

$$A_{\text{dst}} \leftarrow (1 - A_{\text{src}}) A_{\text{dst}} \quad . \tag{10}$$

Here, A_{dst} is the destination transparency that is iteratively obtained and has an initial value of $A_{\text{dst}} = 1$. Ray tracing implicitly returns the correct order of the intersection points, because each intersection is at a certain ray distance. However, traversal of tree-based spatial data structures like BVHs does not always return intersections in the correct order if the tree nodes overlap. Amstutz et al. [AGGW15] found that the correct order of intersections can be quickly obtained because the intersections requiring re-sorting are encountered closely together. Only few operations are required to correct the order of the intersection points.

3. Ray Tracing the Semi-Transparent SES

The rendering of the SES is realized in the ray tracing engine OSPRay. We leveraged the modular structure of OSPRay to implement the missing required primitives and other custom geometries. Key parts of the implementation are intersection, shading and bounding box computation. One of the contributions of our approach is an efficient method for removing the inner parts of tori and spheres to achieve artifact-free rendering of transparent SES.

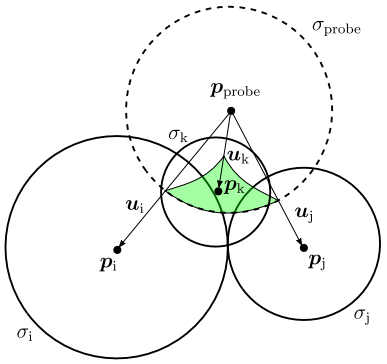

Figure 3: *Depiction of the cutting planes generation for a spherical triangle patch. In green the spherical triangle patch is shown. The three molecule spheres* σ_{i}, σ_{j}, σ_{k}, *and the probe sphere* σ_{probe} *are required for the cutting plane calculation.*

Additionally, a molecular data set contains color information for the individual atoms. This color is then interpolated over the surface during the shading step.

3.1. Cutting Geometries

Since the SES is built from just patches of geometric primitives, parts of each (e. g., a full torus) has to be cut away. In case of a spherical triangle the cutting geometries are planes ($\boldsymbol{P}\boldsymbol{n} = d$, *Hesse normal form*). The probe sphere that is in contact with three spheres at the same time is trimmed by three planes. Therefore, the direction to each neighboring sphere center ($\boldsymbol{p}_{\text{j}}$, $\boldsymbol{p}_{\text{k}}$ and, $\boldsymbol{p}_{\text{probe}}$) is calculated. The directions $\boldsymbol{u}_{\text{i}}$, $\boldsymbol{u}_{\text{j}}$ and, $\boldsymbol{u}_{\text{k}}$ span the planes via the normalized cross product of all combinations. A sketch of the sphere setup and the vectors used for the cutting planes is shown in Figure 3. Two spherical triangles can intersect each other and if not handled correctly this results in visible artifacts. By using the probe sphere as cutting geometry after finding all possible intersecting probe positions for each spherical triangle patch, these singularities can be handled. To find intersecting probe spheres more efficiently than iterating over all spheres, a neighborhood computation on a grid with cell length $\max(1, 2R_{\text{probe}})$ is performed.

To find the correct cutting geometry for the toroidal patch is more involved and requires at least one sphere. The so-called *visibility sphere* [KBE09] is used for clipping the initial torus at circle c_{j} between σ_{i} and σ_{j}. The position of the visibility sphere yields

$$\boldsymbol{p}_{\text{vs}} = \boldsymbol{d} + \boldsymbol{p}_{\text{i}} - \boldsymbol{p}_{\text{probe}} \quad , \tag{11}$$

with

$$\boldsymbol{d} = \frac{||\boldsymbol{p}_{\text{probe}} - \boldsymbol{p}_{\text{i}}|| \, (\boldsymbol{p}_{\text{j}} - \boldsymbol{p}_{\text{i}})}{||\boldsymbol{p}_{\text{probe}} - \boldsymbol{p}_{\text{j}}|| + ||\boldsymbol{p}_{\text{probe}} - \boldsymbol{p}_{\text{i}}||} \tag{12}$$

as the position vector of the visibility sphere relative to $\boldsymbol{p}_{\text{i}}$. The radius of the visibility sphere results to

$$R_{\text{vs}} = \left\| \frac{\boldsymbol{p}_{\text{probe}} - \boldsymbol{p}_{\text{i}}}{\left\| \boldsymbol{p}_{\text{probe}} - \boldsymbol{p}_{\text{i}} \right\|} R_{\text{i}} - \boldsymbol{d} \right\| \quad . \tag{13}$$

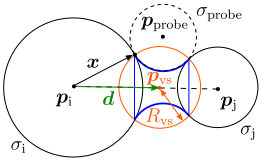

Figure 4: *Schematic of a visibility sphere used for cutting to extract a toroidal patch. The visibility sphere is colored in orange and the vector in green ($\boldsymbol{d}$) points at the center of the visibility sphere relative to the sphere center $\boldsymbol{p}_{\text{i}}$.*

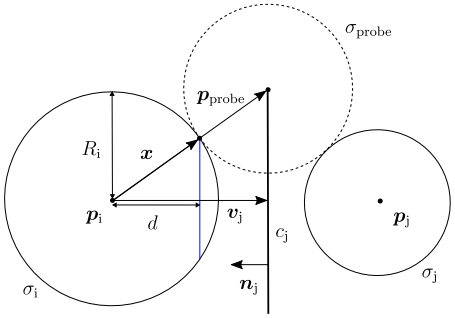

Figure 5: *Schematic of the circle-plane-based cutting approach. Sphere σ_{i} and sphere σ_{j} define a circle c_{j} by intersecting the spheres with their extended radius. The plane this circle defines by its normal is shifted towards the center $\boldsymbol{p}_{\text{i}}$ at distance d. This transformed plane is then used for cutting geometry of sphere σ_{i} away that would lead to artifacts in semi-transparent rendering of the SES.*

Figure 4 depicts the cutting of a torus using the visibility sphere to generate a toroidal patch. For semi-transparent rendering of the SES we need to add two planes that cut the toroidal patch at the arc endpoints $\boldsymbol{s}(a)$ and $\boldsymbol{e}(a)$ (see Section 2.2). The position of the spheres σ_{i} and σ_{j}, and the arc endpoint positions $\boldsymbol{s}(a)$ and $\boldsymbol{e}(a)$ are transformed to local coordinates of the toroidal patch center. Therefore, the resulting transformed sphere positions are denoted as $\boldsymbol{o}_{\text{i}}$ and $\boldsymbol{o}_{\text{j}}$, and the transformed arc positions are called $\boldsymbol{q}_{\text{s}}$ and $\boldsymbol{q}_{\text{e}}$. With the normals at the arc endpoints $\boldsymbol{n}_{\text{s}}$ and $\boldsymbol{n}_{\text{e}}$ the cutting planes are spanned

$$d_{\text{s}} = (\boldsymbol{q}_{\text{e}} - \boldsymbol{q}_{\text{s}})\,\boldsymbol{n}_{\text{s}} \quad , \tag{14}$$

$$d_{\text{e}} = (\boldsymbol{q}_{\text{s}} - \boldsymbol{q}_{\text{e}})\,\boldsymbol{n}_{\text{e}} \quad . \tag{15}$$

Note that the direction the normals are pointing at depends on the angle's sign the arc is defined over. This information is crucial for cutting arcs with angles larger than π.

Intersecting a ray with a toroidal patch can yield up to four intersection points. The solving of these quartics was performed by three different methods:

1. Stabilized Ferrari algorithm (analytical, used by Krone et al. [KBE09])
2. Bairstow method (iterative, see [PTVF07])
3. Sphere tracing (iterative, used by Lindow et al. [LBPH10])

All four intersections are required for semi-transparent rendering. The implementation of the stabilized Ferrari algorithm was ported from Herbison-Evans [HE95]. The ray's first intersection with the geometry is transformed in a way that the torus is centered at the origin and parallel to the *xy*-plane. To increase numerical precision, the origin of the viewing ray is translated close to the torus. This is done using the method proposed by Hart [Har96]. We observed the stabilized Ferrari algorithm to sometimes fail the intersection test, which leads to fully transparent stripes inside a torus patch. Krone et al. [KBE09] used two intersection points for their opaque rendering method, therefore no numerical issues were observed. However, Jurčík et al. [JPSK16] report the same instabilities.

Hence, we tried iterative schemes to reduce artifacts. De Toledo et al. [dTLP07] found the *Newton-Raphson* method best working to find the first intersection. As a starting point the first intersection with the bounding geometry is taken. However, iterating the internal intersection points becomes difficult as they can be distributed arbitrarily. This can lead to a convergence to intersection points that had already been found. The Bairstow method behaves in a way that it immediately gives all four roots on a successful convergence. However, due to polynomial deflation this method becomes numerically unstable [PTVF07].

Thus we decided upon using sphere tracing. The sphere tracing method introduced by Hart [Har96] works reasonably well for finding the first intersection of tori [dTLP07] and toroidal patches [LBPH10], and we extend this method for transparent rendering. Sphere tracing stops if the distance to the surface drops below the ε-Region. Performing further steps does not work, because the method is stuck in this ε-Region and single large steps to escape the Region does not work in situations where the ray is roughly parallel to the surface. However performing multiple constant-sized steps until the ε-Region is passed and then continuing with the standard sphere tracing approach to find the next intersection does not produce any artifacts like the other methods. Additionally, we use the visibility sphere to restrict the area where the algorithm is iterating. The ray is intersected with the visibility sphere and this intersection point is used as a starting point for the sphere tracing. As soon as the ray leaves the visibility sphere, the sphere tracing iteration is stopped.

Convex spherical patches without the requirement of semi-transparency can be rendered as simple spheres. Again the visibility sphere can be utilized for cutting parts out of the convex spherical patches. However, in some cases this is not a sufficient operation, because not all geometry is removed correctly. For example, if three visibility spheres are close together, a small spherical triangle could remain. We developed an improved approach for a correct clipping of the hidden region of the spheres.

To cut out the inner torus regions, we apply a circle-plane-based, therefore we again need the arcs that are used for the toroidal patch construction. The neighbor of sphere σ_{i} is σ_{j} and provides an intersection circle c by their extended radii. The construction of the toroidal patch leads to contact circles of spheres σ_{i} and σ_{j} (see Figure 5). These circles are used to compute a cutting plane that removes the remaining parts of the sphere. This process is performed for all intersecting spheres, except for fully covered spheres, as these regions are already cut by the spheres covering them. For computation we project the tangent point, where sphere σ_{i} and the

probe sphere touch each other, onto the circle normal. The normal $\boldsymbol{n}$ of the cutting plane is not influenced by this step and equals the circle normal $\boldsymbol{n}_\mathrm{j}$. However, the distance d is

$$d = \boldsymbol{n}\boldsymbol{x} \quad , \tag{16}$$

with

$$\boldsymbol{x} = \frac{\boldsymbol{p}_\mathrm{probe}}{\left\|\boldsymbol{p}_\mathrm{probe}\right\|} R_\mathrm{i} \quad . \tag{17}$$

The probe position $\boldsymbol{p}_\mathrm{probe}$ is computed using a vector $\boldsymbol{u}$ that is perpendicular to the circle normal $\boldsymbol{n}_\mathrm{j}$

$$\boldsymbol{p}_\mathrm{probe} = \boldsymbol{v}_\mathrm{j} + r(c_\mathrm{j}) \frac{\boldsymbol{n}_\mathrm{j}}{\left\|\boldsymbol{n}_\mathrm{j}\right\|} \quad , \tag{18}$$

where $\boldsymbol{v}_\mathrm{j}$ is the position vector to the circle center of circle c_j. This approach is more stable than the ray-triangle-based approach, where small triangles can lead to numerical issues in rare cases. Hence, this method is used for our semi-transparent rendering of the SES.

3.2. Bounding Geometry

The calculation of tight-fitting axis-aligned bounding boxes is crucial for the rendering performance of OSPRay. Too large bounding boxes result in additional, unnecessary intersection tests. Our approach of extracting the optimum bounding box depends on the geometry type. For each geometry patch type we generate a set of convex hull points that can be translated into an axis-aligned bounding box. This axis-aligned bounding box is then passed to Embree for the construction of the BVH for fast ray traversal.

In the case of a non-intersecting sphere, the bounding box of this convex spherical patch is defined as a box around the sphere with two times the radius of the sphere as edge length. However, in general a convex spherical patch is a complex construct of a sphere and cutting planes (see Figure 6, left). In fact, the cutting planes are used to reduce the size of the bounding box of the visibility sphere. To be able to shrink the initial bounding box with a cutting plane, the plane should only cut through a pair of opposing faces of the initial box of the whole sphere. If the plane only cuts through opposing faces, the smallest distance of a cut edge to a corner is used to shrink the box. Since the final bounding box has to be axis-aligned, only the maximum distanced point is used for the bounding box construction.

A spherical triangle is a sphere cut by planes (see Figure 6, right). So, a good choice for the initial bounding box is the box of the probe sphere. However, depending on the configuration the spherical triangle is placed and cut, more than half of the bounding box is not filled with geometry. In this case we first extract a tight-fitting bounding box and then obtain the axis-aligned box from corner positions of the tight-fitting bounding box. Therefore, a plane is defined by the three tangent points where the probe sphere touches the surrounding spheres. This is followed by the construction of a circle that is defined by intersecting the plane with the probe sphere. The first four corner points are ordered to an enclosing square around this circle. The next four corners are then obtained by shifting the first points in the direction of the circle normal by the height of the spherical triangle. In the last step, an axis-aligned bounding box is constructed from the resulting eight points. This is similar to the method presented by Lindow et al. [LBPH10].

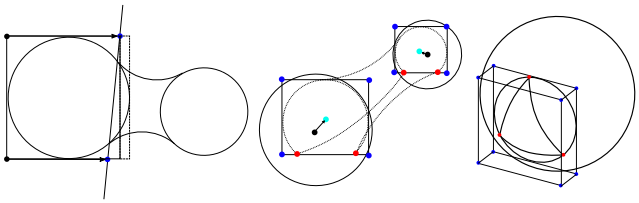

Figure 6: *Depiction of axis-aligned, tight-fitting bounding boxes for the three geometry types. In the left picture the bounding box of a sphere is reduced by the cutting plane of the adjacent torus. Here, blue points show the cutting points with the cube's edges and the arrows suggest the distances used for the size shift. The picture in the center shows a toroidal patch bounding box. The connection arcs of the toroidal patch (endpoints in red) are used to span the two rectangles (corners in blue) that define the bounding box. In the right picture the bounding box of the spherical triangle is depicted (corners in blue). The three tangent points are shown in red.*

The toroidal patch consists of the visibility sphere and two cutting planes (see Figure 6, center). Two cases occur if the arc spans an angle larger than π (first case) or not (second case). In the first case, the intersection circles of the visibility sphere and both neighboring spheres are computed. The *patch corners*, where the toroidal patch connects with neighboring spherical triangles, are calculated. Together with the intersection circles of the visibility sphere, the corners of a bounding rectangle are obtained. With these two rectangles that contain the connection arcs of the toroidal patch to the connected spheres, we derive the axis-aligned bounding box. For the second case, this method is not sufficient, because the toroidal patch can potentially stick out of the bounding box. With α as the angle that the arc is covering, we obtain the lowest point of the toroidal patch at $\frac{1}{2}\alpha$. This lowest point is then used to translate the corners obtained by the first method so that all geometry is covered by the bounding box.

4. Interactive Cavity Highlighting using Ambient Occlusion

Apart from rendering algebraic surfaces and ensuring high image quality our approach is useful for another important use case of interactive ray tracing: visualization of molecular cavities. Similar approaches are commonly found in the literature (see STAR from Krone et al. [KKL*16]). However, these methods are either approximating the cavity extraction or are not interactive. Our semi-transparent rendering method follows the idea of Borland et al. [Bor11]. AO weights are mapped to the opacity of the SES, leaving areas with cavities less transparent than exposed areas. In contrast to Borland et al. [Bor11] our extraction method does not use precomputed AO values that are obtained from a triangulated version of the SES. The method presented here is using real-time occlusion weights that are obtained during rendering. This also affects the image during progressive refinement (implemented via an accumulation buffer in OSPRay): more samples per

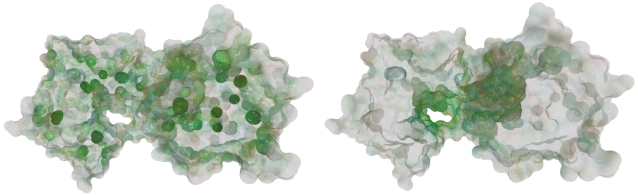

Figure 7: *Shown is the SES of the molecule* 4dfr *using ambient occlusion opacity mapping. This molecule contains several small cavities and a tunnel. On the left side Equation (21) is used to visualize all the small cavities of the molecule. Additionally, the maximum AO sample distance is* 5 Å *and the parameter* $\rho = 1$ *gives a linear scaling. In the right picture Equation (22) is utilized for visualization, the maximum AO sample distance is* 15 Å *and* $\rho = 4$. *Both images use* $\tau = 0.5$.

pixel are increasing the image quality to obtain noiseless renderings.

The ray traced ambient occlusion estimates the AO term O_p at point p by Monte Carlo sampling [PJH16] the surrounding hemisphere [Bor11]

$$O_p = \frac{1}{\pi} \int_\Omega \boldsymbol{n} \cdot \boldsymbol{\omega} V_p(\boldsymbol{\omega}) \mathrm{d}\omega \quad . \tag{19}$$

Here, the surface normal is denoted as $\boldsymbol{n}$ and the AO visibility function is given by

$$V_p(\boldsymbol{\omega}) = \min\left(1, \max(0, \chi(\boldsymbol{\omega})^\rho)\right) \quad , \tag{20}$$

with

$$\chi(\boldsymbol{\omega}) = \frac{1}{\tau}\left(1 - \frac{D_p(\boldsymbol{\omega})}{D_{\max}}\right) \quad , \tag{21}$$

and $D_p(\boldsymbol{\omega})$ as the AO sample distance at point p, hence $D_{\max}$ is the maximum sampling distance. The parameter τ ($0 < \tau \leq 1$) is an opacity threshold that allows faster saturation and acts as a scaling parameter [Bor11]. An additional parameter ρ adjusts for a nonlinear opacity. By taking a closer look at $\chi(\boldsymbol{\omega})$, this function returns high values for close surfaces and leads to heavy occlusion of small cavities regardless of $D_{\max}$. Therefore, an alternative function is provided for the detection of larger cavities

$$\chi(\boldsymbol{\omega}) = \frac{1}{\tau}\left(\frac{D_p(\boldsymbol{\omega})}{D_{\max}}\right) \quad . \tag{22}$$

This AO weighting scheme is inspired by the work of Borland [Bor11]. In Figure 7 an ambient occlusion to opacity mapping using Equation (21) and Equation (22) is shown.

Our coloring approach uses the front to back blending scheme at each intersection and also includes the AO term

$$C_{\mathrm{src}} = \beta O_p C_{\mathrm{blend}} + (1 - \beta O_p) C_{\mathrm{surface}} \quad , \tag{23}$$

where C_{blend} is the color indicating a cavity (user defined), β is a parameter to adjust the influence of the user defined color, and C_{surface} is the surface color returned by the interpolation scheme of Krone et al. [KBE09]. Additionally, we found when using the front to back blending (equation (9)) the AO term O_p will occur squared in the resulting equation, because the color now also contains the term O_p. However, O_p is a nonlinear operator (as used in the original opacity function by Borland et al. [Bor11]) and can not simply be squared, otherwise the resulting images would be biased. In a per ray approach, nonlinear operations on single values result in a completely different behavior than a per pixel approach. For unbiased results we modified this approach and perform the blending operation twice for each intersection point. This mimics the situation of intersecting two surfaces that are infinitesimally close together. In the first blending operation the color is calculated as

$$C_{\mathrm{src}} = (1 - \beta) C_{\mathrm{surface}} + \beta C_{\mathrm{blend}} \quad ,$$

and the source opacity is computed as $A_{\mathrm{src}} = O_p$. The blending for the second surface is performed as

$$\begin{aligned} C_{\mathrm{dst}} &\leftarrow A_{\mathrm{src}} A_{\mathrm{dst}} C_{\mathrm{src}} + C_{\mathrm{dst}} \quad , \\ A_{\mathrm{dst}} &\leftarrow (1 - A_{\mathrm{src}}) A_{\mathrm{dst}} \quad , \\ C_{\mathrm{dst}} &\leftarrow \alpha A_{\mathrm{dst}} C_{\mathrm{surface}} + C_{\mathrm{dst}} \quad , \\ A_{\mathrm{dst}} &\leftarrow (1 - \alpha) A_{\mathrm{dst}} \quad , \end{aligned}$$

where α sets the general opacity of the surface. If there is no additional occlusion by a cavity, the opacity of the surface is still α. For regions with a high AO term O_p the opacity of the surface is increased. In Figure 8 the influence of the parameters α and β is depicted. Shrinking parameter α reduces the opacity of regions without occlusion and increasing parameter β blends more occlusion color into the cavities of the SES. If $\alpha = 1$ and $\beta = 1$ the SES is fully opaque and the AO colors occluded regions in the corresponding occlusion color C_{blend}. Additionally, the visualization remains stable for different numbers of sample rays and does therefore not bias the SES during accumulation.

5. Results and Discussion

We tested the SES generation, transparency rendering, and ambient occlusion opacity mapping rendering performance of commonly used molecules like *1vis*, *1aon*, and *3g71* [KGE11, LBPH10]. Additionally, we added the largest available molecules found in molecule databases to showcase that our CPU implementation is not limited to small memory footprints like GPU implementations. The machine for these performance tests was equipped with an Intel i9-7900x CPU (10 cores), 64 GB of RAM, and an Nvidia Titan Xp GPU. In Figure 9 a visualization of all molecules that were used for performance tests is shown. The results of the SES computation and rendering performance are summarized in Table 2. The first information under the *Count* header displays the number of atoms the data set contains and also details about the contour-buildup algorithm with regard to geometry count and cutting geometry count. In average there are 34 cutting planes per convex sphere patch, which corresponds to the expected number of neighbors per sphere.

The contour-buildup implementation of Krone et al. [KGE11] that runs on a GPU was also measured on the same hardware and compared to our CPU implementation. This is shown in Table 2 under the header *SES Computation*. As expected, the highly parallel architecture of the hardware is roughly an order of magnitude faster than our CPU approach. The implementation of Krone et

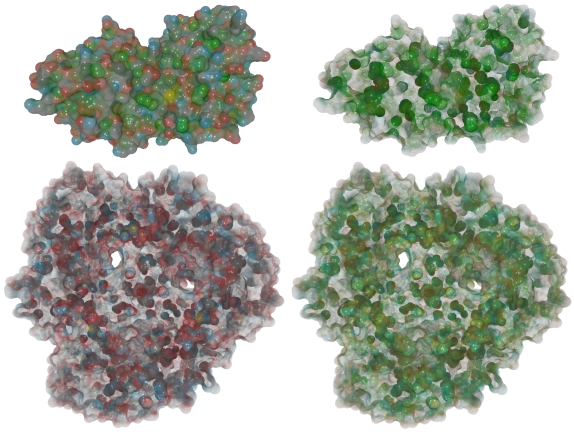

Figure 8: *Visualization of the semi-transparent SES with different combinations of the blending parameters* α *and* β*. The molecule in the top row is* 1vis *and only parameter* α *is varied. The bottom row shows the molecule* 1af6 *for two different values of* β*. The top left picture shows the SES with* $\alpha = 0.5$ *and* $\beta = 1$*. In the top right corner the parameters are* $\alpha = 0.1$ *and* $\beta = 1$*. On the bottom left the parameters are* $\alpha = 0.1$ *and* $\beta = 0$*. The bottom right SES visualization shows* $\alpha = 0.1$ *and* $\beta = 0.5$*. Additionally we used a fixed sample rate (64 samples per pixel) for the* 1vis *molecule and an accumulation over multiple renderings for the* 1af6 *molecule. The accumulated pictures are obtained after several seconds of accumulation to be able to acquire a decent quality.*

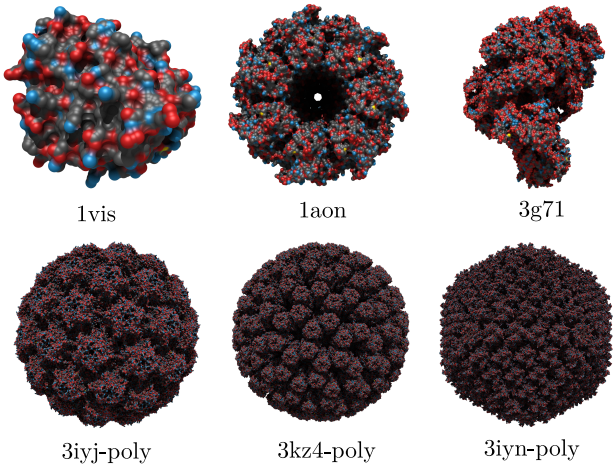

Figure 9: *Depiction of all molecules used for performance tests. These visualizations are using our implementation to generate the SES and were rendered by the OSPRay* scivis *renderer.*

al. [KGE11] can not render transparency and is not dealing with the additional clipping approaches presented in Section 3.1.

Under the header *Rendering* we compared the Ferrari and the sphere tracing intersection algorithms. In previous publications [KGE11] the Ferrari algorithm was chosen because of its performance advantage. Because of numerical issues the Ferrari method however produces visible artifacts in the visualization and should be avoided in situations that are not performance critical.

For semi-transparent renderings, a ray cannot simply be terminated at the first intersection point. Therefore, the rendering performance will be reduced – in some cases drastically. Additionally, the acceleration of the spatial data structure, in our case Embree's BVH, is also reduced. The AOOM also generates occlusion rays at each intersected surface that decrease the performance even further. In Table 3 the results of our performance measuring of the transparency rendering and the AOOM rendering are shown. These results show that both techniques run on interactive frame rates. However, for larger data sets the frames per second dropped to a non-interactive level.

5.1. Instancing

Large molecule complexes typically consist of an assessable amount of different molecule types, but hundreds of thousands of occurrences in the data. This is a well-known problem in computer graphics and is solved by the introduction of instances. An instance simply transforms a base geometry and most importantly does not copy the geometry. The ray tracing engine OSPRay has built-in support for instancing of geometry. This is of special interest for molecular visualizations because larger molecule complexes (e. g. cells) usually contain the same molecule multiple times. This means a SES needs only be computed once and can then be replicated millions of times without memory overhead. To showcase a rendering of many instanced molecules, we used the data set *1af6* and replicated it on a cubic grid with fixed distance between the molecules (no overlapping). The performance over a large range of instance counts is plotted in Figure 10. The clearly visible performance drop for the first 10^3 instances is due to the large amount of intersection computations during ray traversal. Additionally, Embree's BVH spatial structure introduces a large amount of overhead and exceeds the memory of our machine for more than 10^7 instances. This could be improved, by using acceleration structures with less overhead (or no overhead) like *P-k-d* trees [WKJ*15]. A downside of the P-k-d trees in regard of our geometric representations used for the SES, are the tight-fitting bounding boxes we presented earlier in this work. The bounding boxes of a P-k-d tree would appear larger than using the BVH structure with our bounding box optimizations, and therefore reduce the rendering performance due an increased number of intersection tests.

To showcase the capabilities of the instancing of our SES geometry, we use the HIV in blood plasma generated by *cellPack* [JAAA*15]. The data set modelling tool cellPack generates biomolecular structures procedurally and thus is able to generate whole cell data sets with millions of molecule instances. Le Muzic et al. [LMAPV15] also used cellPack to generate data for their level-of-detail GPU rendering tool cellView. The HIV with blood plasma data set contains 40 different types of molecules that are 20.5 k times instanced in total. This results in 61.8 M atoms that adds up, after processing each molecule with the contour-buildup algorithm and considering the number of instances, to 268.1 M primitives. The time to construct the SES of all involved molecules was 0.604 s and consuming 714 MB of render data memory. The

Table 2: *Summarized results of the CPU implementation of the SES computation and rendering. The number of atoms for each molecule, the total number of generated geometries (convex sphere patches, torus patches, and concave sphere patches), and the number of generated cutting geometry for semi-transparent rendering are found under the* Count *header. The next header shows the SES computation times (CB and render data generation) for our CPU implementation and for the GPU implementation of Krone et al. [KGE11]. Differences of tested tori rendering strategies are shown under the* Rendering *header. Additionally, the last two column (*Memory*) shows the memory consumption of our approach of the corresponding data set. Circles and arcs are used to generate the SES, while only the SES render data is kept for rendering. The probe radius for all tests was* $R_{\text{probe}} = 1.4\,\text{Å}$.

	Count			SES Computation [s]		Rendering [fps]		Memory [MB]	
Molecule	Atoms	Geometry	Cutting	GPU	CPU	Ferrari	Sphere Tracing	Rendering	Arcs & Circles
1vis	2.5 k	9.5 k	46.8 k	0.0085	0.0117	100.6	74.2	3.3	15.0
1aon	58.7 k	266.8 k	1.3 M	0.0252	0.2135	71.2	48.8	91.8	346.5
3g71	91.4 k	376.7 k	1.9 M	-	0.3403	87.6	59.5	130.6	546.0
3iyj-poly	1.35 M	6.6 M	32.0 M	-	5.0812	26.9	23.1	2312.8	$8.0 \cdot 10^3$
3kz4-poly	3.24 M	15.0 M	72.4 M	-	12.0472	31.2	20.6	3953.7	$19.2 \cdot 10^4$
3iyn-poly	5.97 M	24,1 M	122.5 M	-	22.2525	26.5	17.7	5186.6	$38.8 \cdot 10^4$

Table 3: *Performance of the transparent and AOOM renderings for a probe radius of* R_{probe}. *The Ferrari algorithm was used for intersection computation. The opacity parameter of the transparency rendering was set to* $\alpha = 0.5$. *For the AOOM, an opacity value* $\alpha = 0.1$ *and a color blending weight of* $\beta = 0.75$ *was set. One AO sample with a maximum distance of* 5 Å *was calculated per frame.*

Molecule	Transparency [fps]	AOOM [fps]
1vis	23.4	9.1
1aon	8.8	2.7
3g71	10.0	2.7

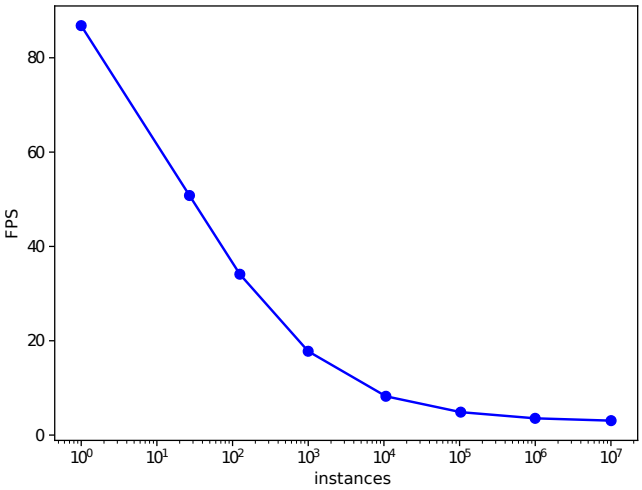

Figure 10: *Rendering performance graph of the molecule* 1af6. *The number of instances was increased approximately by an order of magnitude in each step. The rendering is using the Ferrari algorithm for ray tracing of the toroidal patches.*

top view of Figure 11 shows the whole HIV with blood plasma data set rendering at 3.8 fps, while on the bottom the front half space is cut away to emphasize the envelope and the capsid of the HIV.

Please note that for the HIV data set, the CellView renderer can maintain a performance of over 50 FPS. One reason for the high performance is its level-of-detail approach, the other is that CellView renders simple sphere primitives in all cases instead of computing the correct SES like our approach does.

As a preview for HPC computation of such data sets, we perform a spacial subdivision of the data, so the parts can be computed on different nodes. We measure the rendering performance for each chunk to get an estimation of the HPC performance of the HIV with blood plasma data set. In Figure 12 the results for subdivisions into 8, 64 and 512 chunks is measured. For 8 and 64 chunks the visualization is already interactive. This small measurement series does not include the overhead an image composition tool like *IceT* [MKPH11] would add, however we can clearly see that we can easily load a much larger data set in the HPC scenario for node numbers larger than 64 and still are able to render interactively.

6. Conclusion and Future Work

We presented a CPU implementation of the SES geometry that uses OSPRay [WJA*17] for computation and rendering high quality images. The visualization is integrated into the framework MegaMol [GKM*15, RKRE17]. Our CPU approach is an alternative to the established GPU approaches for rendering the SES of molecular complexes [KGE11, LMAPV15]. However, our CPU implementation can be utilized in situ on any HPC system as it does not rely on the availability of GPUs. We used the contour-buildup algorithm to obtain the algebraic geometry of the SES and provided optimizations to the CSG-based cutting geometry algorithms that are used to obtain artifact-free semi-transparent renderings. We used an AOOM rendering technique for interactive visual detection and inspection of cavities [Bor11]. It is possible to switch between two different visibility functions what allows for adjusting to a desired cavity granularity.

Our performance tests show that we are not able to beat the per-

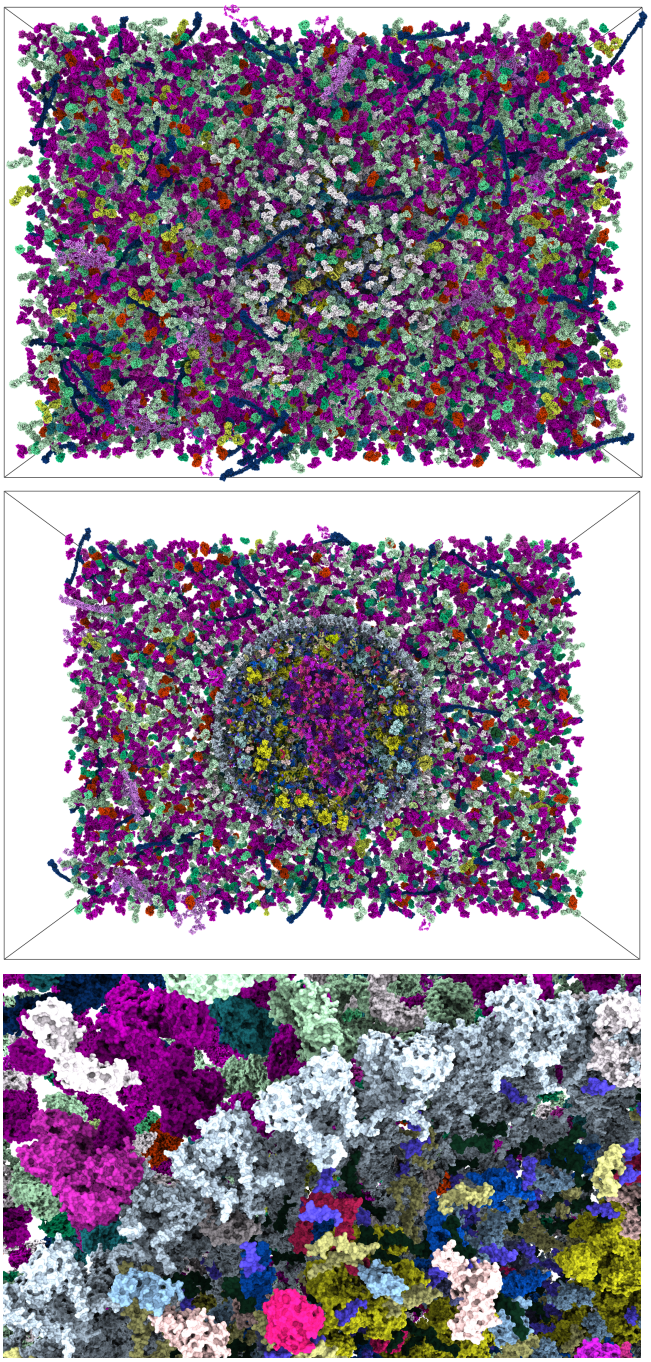

Figure 11: *Rendering of the HIV in blood plasma data set at 3.8 fps. The top scene contains 20.5 k instances of 40 molecule types. The total 61.8 M atoms are represented by a total of 268.1 M primitives. All instances of a molecule are rendered in the same color. The middle rendering shows the same data set cut in half so the HIV capsid and envelope are visible. In the bottom image, a close-up view unveils the details of the SES.*

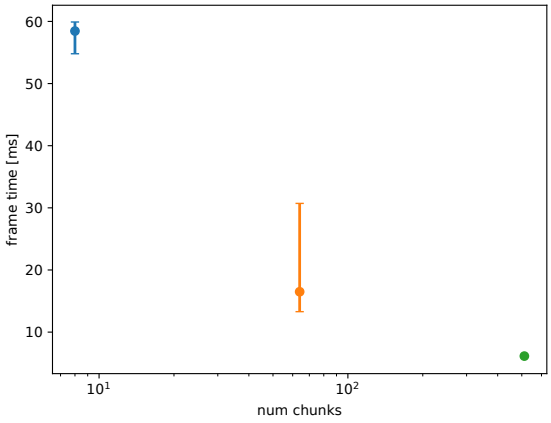

Figure 12: *Performance plot of the three different subdivision schemes (8, 64 and 512). Each chunk was rendered and the time to calculate a frame was measured. The graph shows the average value as a dot, and the minimum and the maximum as an error bar.*

formance of the GPU implementation of Krone et al. [KGE11]. Future performance optimizations could be realized considering vectorization of our code. We already attempted a naive vectorization (SIMDization) of our algorithms, however the performance achieved was not satisfactory. By reducing the branching of the CB algorithm the implementation could benefit from vectorization. Also the computation of the surface itself requires several seconds on up-to-date hardware, this limits the interactivity of time-dependent renderings of the SES with our approach. This problem can, however, be resolved by parallelizing the SES calculation over multiple machines, which is now straightforward with our implementation. Further, larger systems, such as the system presented by Le Muzic [LMPSV14], could be rendered interactively as our results from the instancing tests are suggesting.

Acknowledgements

This research was partially supported by the Intel® Graphics and Visualization Institutes of XeLLENCE program and by the German Research Foundation (DFG) as part of SFB 716 projects D.3 and D.4.

References

[AGGW15] AMSTUTZ J., GRIBBLE C., GÜNTHER J., WALD I.: An evaluation of multi-hit ray traversal in a bvh using existing first-hit/any-hit kernels. *Journal of Computer Graphics Techniques (JCGT) 4*, 4 (2015). 5

[BM08] BAVOIL L., MYERS K.: Order independent transparency with dual depth peeling. *NVIDIA OpenGL SDK* (2008), 1–12. 5

[Bor11] BORLAND D.: Ambient Occlusion Opacity Mapping for Visualization of Internal Molecular Structure. *Journal of WSCG 19*, 1 (2011), 17–24. 2, 7, 8, 10

[Bru19] BRUCKNER S.: Dynamic Visibility-Driven Molecular Surfaces. *Computer Graphics Forum 38*, 2 (May 2019), 317–329. 4

[CCW06] CAN T., CHEN C.-I., WANG Y.-F.: Efficient molecular surface generation using level-set methods. *Journal of Molecular Graphics & Modelling 25*, 4 (Dec. 2006), 442–454. 3

[Con83] CONNOLLY M. L.: Analytical molecular surface calculation. *Journal of applied crystallography 16*, 5 (1983), 548–558. 2, 3

[dTLP07] DE TOLEDO R., LEVY B., PAUL J.-C.: Iterative Methods for Visualization of Implicit Surfaces On GPU. In *Advances in Visual Computing* (2007), Bebis G., Boyle R., Parvin B., Koracin D., Paragios N., Tanveer S.-M., Ju T., Liu Z., Coquillart S., Cruz-Neira C., Müller T., Malzbender T., (Eds.), Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 598–609. 6

[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. 4

[GKM*15] GROTTEL S., KRONE M., MÜLLER C., REINA G., ERTL T.: MegaMol – Prototyping Framework for Particle-Based Visualization. *IEEE Transactions on Visualization and Computer Graphics 21*, 2 (Feb. 2015), 201–214. 5, 10

[Gum03] GUMHOLD S.: Splatting Illuminated Ellipsoids with Depth Correction. In *Vision, Modeling, and Visualization* (2003), pp. 245–252. 4

[Har96] HART J. C.: Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*, 10 (Dec. 1996), 527–545. 6

[HDS96] HUMPHREY W., DALKE A., SCHULTEN K.: VMD – Visual Molecular Dynamics. *Journal of Molecular Graphics 14* (1996), 33–38. 4

[HE95] HERBISON-EVANS D.: I.1 - Solving Quartics and Cubics for Graphics. In *Graphics Gems V*, Paeth A. W., (Ed.). Academic Press, Boston, 1995, pp. 3–15. 6

[HKG*17] HERMOSILLA P., KRONE M., GUALLAR V., VÁZQUEZ P.-P., VINACUA L., ROPINSKI T.: Interactive GPU-based generation of solvent-excluded surfaces. *The Visual Computer 33*, 6 (2017), 869–881. 3

[JAAA*15] JOHNSON G. T., AUTIN L., AL-ALUSI M., GOODSELL D. S., SANNER M. F., OLSON A. J.: cellPACK: a virtual mesoscope to model and visualize structural systems biology. *Nature Methods 12*, 1 (Jan. 2015), 85–91. 9

[JPSK16] JURCIK A., PARULEK J., SOCHOR J., KOZLIKOVA B.: Accelerated Visualization of Transparent Molecular Surfaces in Molecular Dynamics. In *IEEE Pacific Visualization Symposium* (2016), pp. 112–119. 4, 6

[KBE09] KRONE M., BIDMON K., ERTL T.: Interactive Visualization of Molecular Surface Dynamics. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1391–1398. 4, 5, 6, 8

[KGE11] KRONE M., GROTTEL S., ERTL T.: Parallel Contour-Buildup Algorithm for the Molecular Surface. In *IEEE Symposium on Biological Data Visualization* (2011), pp. 17–22. 1, 3, 4, 8, 9, 10, 11

[KKF*17] KOZLÍKOVÁ 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 36*, 8 (2017), 178–204. 1, 2, 3, 4

[KKL*16] KRONE M., KOZLÍKOVÁ B., LINDOW N., BAADEN M., BAUM D., PARULEK J., HEGE H.-C., VIOLA I.: Visual Analysis of Biomolecular Cavities: State of the Art. *Computer Graphics Forum 35*, 3 (2016), 527–551. 5, 7

[KKP*13] KAUKER D., KRONE M., PANAGIOTIDIS A., REINA G., ERTL T.: Rendering molecular surfaces using order-independent transparency. In *EGPGV* (2013), pp. 33–40. 4

[LBH12] LINDOW N., BAUM D., HEGE H.-C.: Interactive Rendering of Materials and Biological Structures on Atomic and Nanoscopic Scale. *Computer Graphics Forum 31*, 3 (2012), 1325–1334. 4

[LBPH10] LINDOW N., BAUM D., PROHASKA S., HEGE H.-C.: Accelerated Visualization of Dynamic Molecular Surfaces. *Computer Graphics Forum 29*, 3 (2010), 943–952. 1, 3, 6, 7, 8

[LMAPV15] LE MUZIC M., AUTIN L., PARULEK J., VIOLA I.: cellVIEW: a Tool for Illustrative and Multi-Scale Rendering of Large Biomolecular Datasets. *Eurographics Workshop on Visual Computing for Biomedicine 2015* (2015), 61–70. 9, 10

[LMPSV14] LE MUZIC M., PARULEK J., STAVRUM A.-K., VIOLA I.: Illustrative visualization of molecular reactions using omniscient intelligence and passive agents. In *Computer Graphics Forum* (2014), vol. 33, Wiley Online Library, pp. 141–150. 11

[MKPH11] MORELAND K., KENDALL W., PETERKA T., HUANG J.: An Image Compositing Solution at Scale. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 25:1–25:10. 10

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: OptiX: A General Purpose Ray Tracing Engine. In *ACM SIGGRAPH 2010 Papers* (New York, NY, USA, 2010), SIGGRAPH '10, ACM, pp. 66:1–66:13. 4

[PJH16] PHARR M., JAKOB W., HUMPHREYS G.: *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016. 8

[PM12] PHARR M., MARK W. R.: ispc: A SPMD compiler for high-performance CPU programming. *2012 Innovative Parallel Computing (InPar)* (2012), 1–13. 5

[PTVF07] PRESS W. H., TEUKOLSKY S. A., VETTERLING W. T., FLANNERY B. P.: *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007. 6

[PV12] PARULEK J., VIOLA I.: Implicit representation of molecular surfaces. In *2012 IEEE Pacific Visualization Symposium* (Feb. 2012), pp. 217–224. 3

[RE05] REINA G., ERTL T.: Hardware-Accelerated Glyphs for Mono- and Dipoles in Molecular Dynamics Visualization. In *EG/IEEE VGTC Symposium on Visualization* (2005), pp. 177–182. 4

[Ric77] RICHARDS F. M.: Areas, volumes, packing, and protein structure. *Annual review of biophysics and bioengineering 6*, 1 (1977), 151–176. 2

[RKRE17] RAU T., KRONE M., REINA G., ERTL T.: Challenges and Opportunities using Software-Defined Visualization in MegaMol. In *Workshop on Visual Analytics, Information Visualization and Scientific Visualization (WVIS) in the 30th Conference on Graphics, Patterns and Images (SIBGRAPI'17)* (Niterói, RJ, Brazil, Oct. 2017), Ferreira N., Nonato L. G., Sadlo F., (Eds.). 5, 10

[SMSS16] STONE J. E., MESSMER P., SISNEROS R., SCHULTEN K.: High Performance Molecular Visualization: In-Situ and Parallel Rendering with EGL. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (May 2016), pp. 1014–1023. 4

[SOS96] SANNER M. F., OLSON A. J., SPEHNER J.-C.: Reduced surface: An efficient way to compute molecular surfaces. *Biopolymers 38*, 3 (Mar. 1996), 305–320. 3, 4

[TA96] TOTROV M., ABAGYAN R.: The Contour-Buildup Algorithm to Calculate the Analytical Molecular Surface. *Journal of Structural Biology 116*, 1 (Jan. 1996), 138–143. 1, 3

[TCM06] TARINI M., CIGNONI P., MONTANI C.: Ambient Occlusion and Edge Cueing for Enhancing Real Time Molecular Visualization. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (Sept. 2006), 1237–1244. 2

[WJA*17] WALD I., JOHNSON G., AMSTUTZ J., BROWNLEE C., KNOLL A., JEFFERS J., GÜNTHER J., NAVRATIL P.: OSPRay - A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (Jan. 2017), 931–940. 2, 4, 10

[WKJ*15] WALD I., KNOLL A., JOHNSON G. P., USHER W., PASCUCCI V., PAPKA M. E.: CPU ray tracing large particle data with balanced P-k-d trees. In *2015 IEEE Scientific Visualization Conference (SciVis)* (2015), IEEE, pp. 57–64. 9

[WMG*09] WALD I., MARK W. R., GÜNTHER J., BOULOS S., IZE T., HUNT W., PARKER S. G., SHIRLEY P.: State of the Art in Ray Tracing Animated Scenes. *Computer Graphics Forum 28*, 6 (Sept. 2009), 1691–1722. 2

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Trans. Graph. 33*, 4 (July 2014), 143:1–143:8. 5

[Zey09] ZEYUN YU: A list-based method for fast generation of molecular surfaces. In *2009 Annual International Conference of the IEEE Engineering in Medicine and Biology Society* (Minneapolis, MN, Sept. 2009), IEEE, pp. 5909–5912. 3