Institute for Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Interactive Ray Tracing of Solvent Excluded Surfaces

Sebastian Zahn

Course of Study:	Master of Science Informatik
Examiner:	Prof. Dr. Thomas Ertl
Supervisor:	Dr. Michael Krone, Tobias Rau, M. Sc.
Commenced:	November 15, 2017
Completed:	May 15, 2018

Abstract

Domain experts in fields concerned with the behavior of molecules, for example biochemists, employ simulations to study a molecule's individual properties and mutual interactions with other molecules. To obtain an intuitive spatial understanding of the returned data of the simulations, various visualization techniques such as molecular surfaces can be applied on the data. The solvent excluded surface depicts the boundary between the molecule's and a solvent's occupied space and therefore the molecules accessibility for the solvent. Insight about a molecule's potential for interaction such as reactions can be gained by studying the surface's shape visually. Current implementations for the visualization of the surface usually utilize GPU ray casting to achieve the performance required to allow interactivity such as viewpoint changing. However, this makes implementation of physically motivated effects like ambient occlusion or global illumination difficult. If compute resources do not contain GPUs, which is often the case in compute clusters, expensive software rasterization has to be employed instead. As CPUs offer less parallelism compared to GPUs, overhead introduced by the overdraw of thousands of primitives should be avoided. To mitigate these issues, CPU visualization approaches resurfaced again in recent times. In this work, the solvent excluded surface is visualized interactively using the classic ray tracing approach within the OSPRay CPU ray tracing framework. The described implementation is able to compute and visualize the solvent excluded surface for datasets composed of millions of atoms. Additionally, the surface supports transparency rendering, which allows implementation of a cavity visualization method that uses ambient occlusion.

Kurzfassung

Experten auf Gebieten welche das Verhalten von Molekülen untersuchen, wie zum Beispiel Biochemiker, verwenden Simulationen um die individualen Eigenschaften von Molekülen, und gegenseite Interaktion zwischen diesen zu untersuchen. Um ein intuitives räumliches Verständnis der Daten zu erhalten, werden verschiedene Visualisierungstechniken, wie beispielsweise molekulare Oberflächen, verwendet. Die Solvent Excluded Surface beschreibt die Grenze zwischen dem Raum welcher durch ein Molekül besetzt ist und dem Raum den ein Lösungsmittel einnehmen kann, und stellt daher die Zugänglichkeit des Moleküls für das Lösungsmittel dar. Durch visuelle Untersuchung dieser Oberfläche können Rückschlüsse auf das mögliche Interaktionspotential, wie zum Beispiel Reaktionen, gezogen werden. In heutigen Implementierungen für die Visualisierung dieser Oberfläche kommt üblicherweise GPU Ray Casting zum Einsatz um die nötige Geschwindigkeit zu erreichen, die Interaktivität wie das Verändern der Kameraposition erlaubt. Solche Techniken erschweren die Implementation von physikalisch motivierten Effekten wie Ambient Occlusion und globale Beleuchtungseffekte. Falls verwendete Rechnerressourcen, etwa in einem Rechnerverbund, über keine GPU verfügen, muss teure Software-Rasterisierung verwendet werden. Da CPUs verglichen mit GPUs über weniger Parralellismus verfügen sollte der Aufwand, der durch Overdraw von tausenden Primitiven entsteht, vermieden werden. Um dieses Problem zu umgehen sind CPU-Visualisierungstechniken in letzer Zeit wieder vermehrt in den Blickpunkt geraten. In dieser Arbeit wird die Solvent Excluded Surface interaktiv durch den klassischen Ray-Tracing-Ansatz innerhalb des OSPRay CPU Ray-Tracing Framworks visualisiert. Die Implementierung ist in der Lage, die Solvent Excluded Surface für Datensätze zu berechnen und zu visualisieren, welche aus Millionen von Atomen bestehen. Zusätzlich unterstützt die Oberfläche Transparenz, was die Implementierung eines Verfahrens erlaubt, das Hohlräume durch das Ambient Occlusion Verfahren visuell darzustellt.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

Molecular dynamic simulations range back up to 60 years, when the first simulations were conducted [AM06]. Research fields that utilize these simulations include biochemistry and biophysics, molecular biology and pharmaceutical areas. Simulation runs return datasets that describe the spatial arrangement of atoms that form a molecule [KKL+15]. Each atom's positions is governed by attracting and repulsive forces depending on existing bonds and atom positions. Researchers studying biomolecules such as proteins are especially interested in the potential of interaction between different molecules. These interactions happen especially in regions called *binding sites* [KKL+16], where molecules have certain geometrical characteristics that allow the forming of bounds. Visualization of these datasets is required to give domain experts the ability to explore the computed results and draw conclusions about the behavior of molecules. Employed visualization techniques need to be interactive to allow exploratory viewing of the data, such as view position changing and zooming.

1.1 Motivation

The *solvent excluded surface* [Con83] is the most popular choice for visualization of the accessibility of a studied molecule for another molecule, approximated as a sphere. This spherical approximation is called *probe*. Intuitively, the surface describes the boundary between the molecule and the space the probe may occupy. As the surface is composed of connected implicit surface patches, ray casting and ray tracing are natural choices as applied rendering techniques.

Recent implementations such as the ones presented by Krone et al. [KBE09] and Lindow et al. [LBPH10] utilize GPU ray casting for rendering. These approaches require the presence of GPUs, which is oftentimes not the case in high performance computing (HPC) clusters. Software rasterization allows emulation of GPUs if required, however the overhead introduced by overdraw of many primitives cannot be mitigated as easily on CPUs, as they provide less parallelism. Therefore, a CPU based solution is required, if software rasterization is to be avoided. The OSPRay framework [WJA+17] implements visualization techniques based on CPU ray tracing. Efficient acceleration structures allow higher performance than software rasterization. The ray tracing aspect simplifies implementation of realistic illumination effects, while still offering interactive frame rates [WMG+07]. The ability to efficiently visualize the solvent excluded surface should be introduced into OSPRay.

1.2 Challenges

In absence of GPUs, the surface has to be computed on the CPU, which requires an efficient parallelized implementation. This implementation should possibly be vectorized to utilize CPU vector units that allow parallel computation of simple operations. The surface computation algorithm should be fully integrated into the OSPRay framework, such that users are able to use the surface in the same way as any other geometry offered by the framework. Users are just required to specify the atom dataset with corresponding atom radii to obtain the surface geometry. Along with the OSPRay integration, the surface should also be made available in the MegaMol framework [GKM+15].

Additionally, the surface's connected primitives should be rendered efficiently and in high quality. This requires computation of tightly-fitting axis aligned bounding boxes and efficient intersection routines. The surface consists in part of quartic surfaces. Intersection of such a surface with a ray is numerically not stable, which may lead to visual artifacts. As the geometry should be usable with any renderer offered by OSPRay, such as the path tracer, the implementation must be general enough to allow callback based intersection with rays. This means that given a ray and a primitive, the correct intersection is computed without any other information, except the data associated with the primitive. Another aspect is the support for transparent surface rendering, which allows users to inspect internal structures of the molecule. Commonly, the surface is visualized opaquely, which means that protruding, but hidden surface parts do not have to be removed, as they are covered by the visible parts of the surface. To avoid visual artifacts, these have to be removed, for example by cutting geometry. The callback based intersection computation discourages a solution that requires global information such as the one proposed by Kauker et al. [KKP+13]. Finally, the generated surface is to be utilized to implement an ambient occlusion based cavity visualization technique. This becomes possible with support for transparent rendering.

2 Fundamentals

This chapter discusses the mathematical fundamentals required for ray tracing the geometric primitives the rendered surface is composed of. First, the concept of the implicit surface is introduced. Next the general approach of finding the intersection between an implicit surface and a ray is described. Further, a detailed derivation of the analytic solution and some iterative approaches for solving general quartic equations are shown, which are possible methods to use for solving certain intersection problems. Ray tracing benefits from the usage of acceleration data structures, which are briefly described as well. The last section discusses the software used in the implementation of the geometry.

2.1 Implicit Surfaces

The molecular surface consists of connected implicit surfaces. Such a surface is described by a implicit function $\phi(\vec{x})$ which separates the space $\mathbb{R}^n$ in subdomains, where $\Omega^- \subset \mathbb{R}^3$ can be called the inside portion and $\Omega^+ \subset \mathbb{R}^3$ the outside portion of the domain, assuming there are only two subdomains [OF03]. Between each region lies the boundary $\partial\Omega$ that separates the subdomains, which is also called *interface*. The boundary is located at all $\vec{x}$ where $\phi(\vec{x})$ vanishes i.e.

$$\partial\Omega = \{\vec{x} | \phi(\vec{x}) = 0\} \tag{2.1}$$

and has dimensionality $n-1$, referring to the number of variables required to specify every boundary point.

Moving from a region where $\phi(\vec{x}) < 0$ to $\phi(\vec{x}) > 0$ (or vice versa) implies crossing of the surface [OF03]. The normal vector $\vec{n}_{\vec{x}}$ at $\vec{x} = (x, y, z)^{\mathrm{T}}$ with $\vec{x} \in \partial\Omega$ is directly computed from the gradient

$$\nabla\phi(\vec{x}) = \begin{pmatrix} \frac{\partial\phi}{\partial x} \\ \frac{\partial\phi}{\partial y} \\ \frac{\partial\phi}{\partial z} \end{pmatrix}. \tag{2.2}$$

The normal $\vec{n}_{\vec{x}}$ at $\vec{x}$ is equivalent to the normalized gradient, i.e.

$$\vec{n}_{\vec{x}} = \frac{\nabla\phi(\vec{x})}{\|\nabla\phi(\vec{x})\|}. \tag{2.3}$$

Inside Ω^-, the normal vector $\vec{n}_{\vec{x}}$ will point towards Ω^+ as it points into the direction of steepest increase of $\phi(\vec{x})$. See Figure 2.1 for a depiction.

Implicit surfaces defined by ϕ_1, and ϕ_2 can be combined to construct new surfaces. For example

$$\phi_3(\vec{x}) = \max(\phi_1(\vec{x}), \phi_2(\vec{x})) \tag{2.4}$$

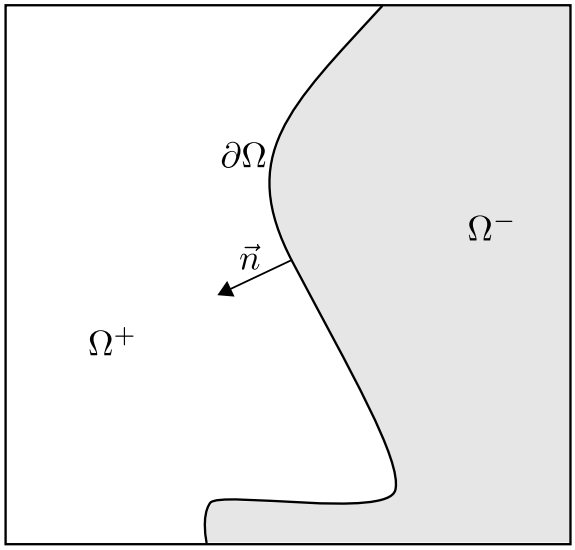

Figure 2.1: Example of an implict surface in $\mathbb{R}^2$. In gray: The Ω^- subdomain where the implicit function is negative. In white: The Ω^+ subdomain where the implicit function is positive.

describes the intersection of ϕ_1, and ϕ_2. This approach is called *constructive solid geometry* (CSG) [OF03; SM09].

In the following $\langle \vec{a}, \vec{b} \rangle$ notates the scalar product between $\vec{a}$ and $\vec{b}$. Rendering of the molecular surface used in this work requires three geometric primitives in $\mathbb{R}^3$:

1. A *sphere* [SM09] centered at $\vec{c}$ with radius R is described by

$$\phi(\vec{x}) = \langle \vec{x} - \vec{c}, \vec{x} - \vec{c} \rangle - R^2. \tag{2.5}$$

2. The *plane* [SM09] defined by a normal $\vec{n}$ and some position vector $\vec{p}$ in the plane is described by

$$\phi(\vec{x}) = \langle \vec{p} - \vec{x}, n \rangle \tag{2.6}$$

3. A *torus* [Har96; KBE09] with minor radius r and major radius R (Figure 2.2) is described by

$$\phi(\vec{x}) = \left(R - \sqrt{x^2 - y^2} \right)^2 + z^2 - r^2 \tag{2.7}$$

2.1.1 Ray-Surface Intersection

One method of rendering images of implicit surface utilizes ray tracing [Gla89]. For every image pixel, a ray is constructed that originates at the eye and passes through this pixel. Each such viewing ray is intersected with the implicit surface. The intersection point of closest distance along the ray is shaded according to the light sources in the scene to obtain the corresponding pixel's color. Figure 2.3 shows a simple example. This basic method can

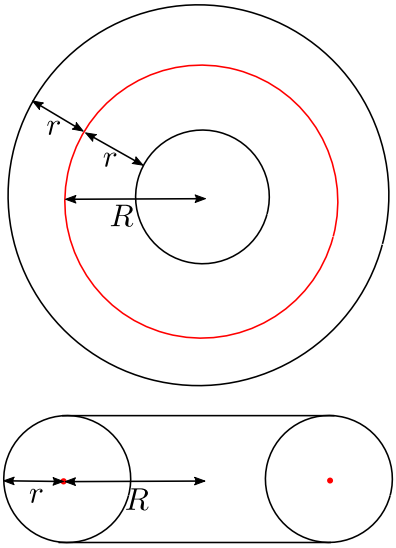

Figure 2.2: Top down and side view depiction of a torus defined by minor radius r and major radius R.

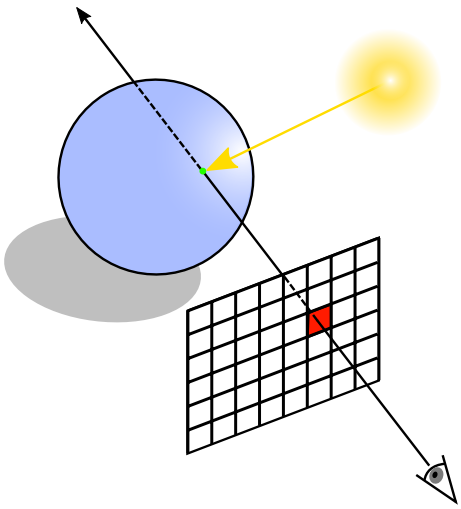

Figure 2.3: Ray tracing of the scene consisting of the blue sphere. The ray is generated for the red pixel and intersected with the sphere to obtain the green intersection point. This point is then shaded according to the light source (yellow) to determine the pixel's color.

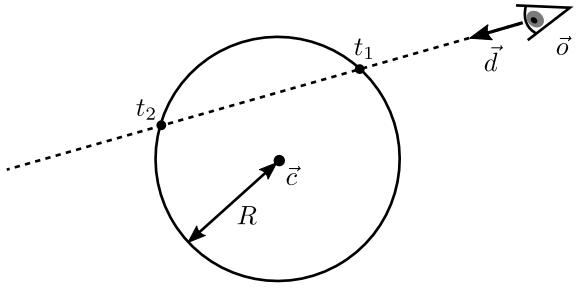

Figure 2.4: Intersection of a ray with a sphere located at $\vec{c}$ with radius R, which yields the intersections at $t_{1,2}$.

be extended to obtain physically realistic images by implementing *path tracing* [Kaj86], where the amount of light arriving at surface points is computed by repeatedly following random paths along rays through the scene. The core mechanism of ray tracing requires the ability to intersect a primitive described by an implicit function with a viewing ray. Let

$$\vec{r}(t) = \vec{o} + t\vec{d} \tag{2.8}$$

be the ray originating at $\vec{o}$ oriented in direction $\vec{d}$. In general, to find the intersection of $\vec{r}(t)$ with the implicit surface $\phi(\vec{x})$, one has to solve the equation

$$\phi(\vec{r}(t)) = 0 \tag{2.9}$$

for t [SM09]. For example, to find the intersection between a sphere defined by Equation 2.5 and a ray $\vec{r}(t)$, the ray is inserted into $\phi(\vec{x})$, yielding

$$0 = \langle \vec{o} + t\vec{d} - \vec{c}, \vec{o} + t\vec{d} - \vec{c} \rangle - R^2 \tag{2.10}$$

$$\iff 0 = \underbrace{\langle \vec{d}, \vec{d} \rangle}_{\alpha} t^2 + \underbrace{2\langle \vec{d}, \vec{o} - \vec{c} \rangle}_{\beta} t + \underbrace{\langle \vec{o} - \vec{c}, \vec{o} - \vec{c} \rangle - R^2}_{\gamma}. \tag{2.11}$$

This is the quadratic equation $\alpha t^2 + \beta t + \gamma = 0$. Therefore, the intersections are at

$$t_{1,2} = \frac{-\beta}{2\alpha} \mp \frac{\sqrt{\beta^2 - 4\alpha\gamma}}{2\alpha}, \tag{2.12}$$

where t_1 describes the entry point and t_2 the exit point. Only the real solution are of relevance here, as complex ones indicate no intersection. If the ray's direction is a unit vector, t corresponds to the distance of intersection from the origin $\vec{o}$. Figure 2.4 depicts an example.

While the closed solution of the ray-sphere intersection problem is straightforward to compute, in general this is not the case for arbitrary implicit surfaces. One such more complex case is the torus, which appears as a primitive in the solvent excluded surface.

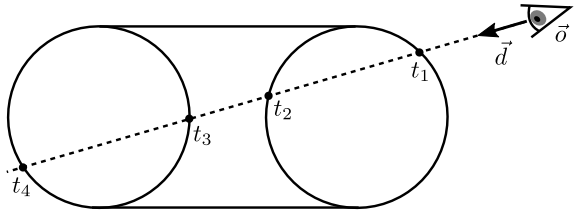

Figure 2.5: Example intersection of a ray with a torus yields the intersections t_1, t_2, t_3 and t_4.

The torus according to Equation 2.7 intersected with the ray $\vec{r}(t)$ yields the quartic [TLP07]

$$t^4 + \alpha t^3 + \beta t^2 + \gamma t + \rho = 0 \tag{2.13}$$

where

$$\begin{aligned}
\alpha &= 4\langle \vec{o}, \vec{d} \rangle, \\
\beta &= 2(\langle \vec{o}, \vec{o} \rangle - (R^2 + r^2) + 2(\langle \vec{o}, \vec{d} \rangle)^2 + 2R^2 d_z^2, \\
\gamma &= 4(\langle \vec{o}, \vec{d} \rangle(\langle \vec{o}, \vec{o} \rangle - (R^2 - r^2)) + 2R^2 d_z o_z), \\
\rho &= (\langle \vec{o}, \vec{o} \rangle - (R^2 + r^2))^2 - 4R^2(r^2 - o_z^2).
\end{aligned}$$

Similar to the ray-sphere intersection problem, finding the real roots of this equation yields the distances along the ray where intersections occur. There are up to four intersections, as shown in Figure 2.5. However, obtaining the solution is not as simple as the quadratic case. Section 2.2 describes analytic and iterative methods to solve this problem.

2.1.2 Sphere Tracing

Some implicit functions $\phi(\vec{x})$ describe the minimal squared distance of $\vec{x}$ to a geometric surface, such as Equation 2.5 and Equation 2.7. Therefore the distance function $a(\vec{x})$ is derived that describes the Euclidean distance to the surface [Har96] as

$$a(\vec{x}) = \min_{y \in \partial\Omega} \|\vec{x} - \vec{y}\|. \tag{2.14}$$

In such a case the sphere tracing method introduced by Hart [Har96] can be employed to find ray-surface intersections. This method is iterative. Consider some position $\vec{x}_t$ and let $a(\vec{x})$ be a distance function that returns Euclidean distance to a surface located at $\phi(\vec{x}) = 0$. Then the implicit function evaluation $a(\vec{x})$ can be used to progressively step along a ray in direction $\vec{d}$ towards the surface with

$$\vec{x}_{t+1} = \vec{x}_t + a(\vec{x})\vec{d}. \tag{2.15}$$

Note that $\vec{d}$ must have unit length. No intersection will be missed as $\vec{x}_{t+1}$ will at most reach exactly the closest surface point, therefore

$$a(\vec{x}_\infty) = 0. \tag{2.16}$$

In practice the iteration can be stopped when the distance drops below a certain ϵ threshold. See Figure 2.6 for an example.

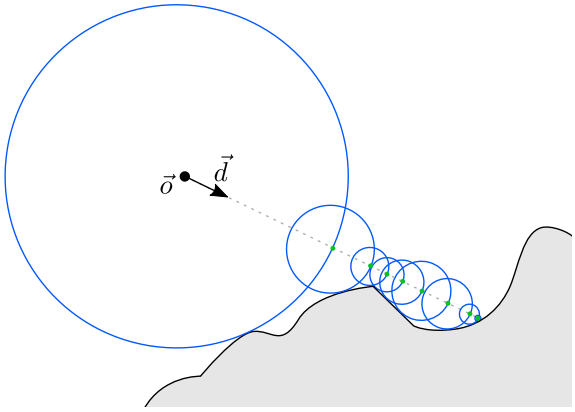

Figure 2.6: Example iteration of sphere tracing, originating at $\vec{o}$ in direction $\vec{d}$. Depicted in blue are the minimum distances returned by the distance function. The green points correspond to the moved positions after each step.

2.2 Quartics

In order to solve the ray-torus intersection problem, the real roots of the quartic equation of Equation 2.13 have to be found. Let

$$x^4 + ax^3 + bx^2 + cx + d = 0 \tag{2.17}$$

be a general quartic equation with real coefficients, i.e. $a, b, c, d \in \mathbb{R}$. Note that any polynomial with non neutral coefficient in front of the x^4 term can be divided out to arrive at this form. See Figure 2.7 for an example.

2.2.1 Analytical Solution

Quartics can still be solved analytically in closed form. Polynomials of higher degree do not allow analytical solving [Fau96]. Different methods exist, however Herbison-Evans found that the Ferrari approach is numerically most stable [Her95].

In the Ferrari approach, the main idea of finding the solutions of Equation 2.17 consists of refactoring it into two quadratic equations. Solving both quadratic equations then yields all four solutions. Arriving at the solution of Equation 2.17 involves constructing and solving a cubic equation called *cubic resolvent* [Her95; Tur57], whose coefficient are computed from the quartic equation's coefficients.

Ferrari's Approach for Quartic Equations

The following derivation generally follows the approach presented by Turnbull [Tur57] to arrive at the equations presented by Herbison-Evans [Her95]. Weisstein [Wei18a] gives another approach to derive a similar cubic resolvent, however their approach is based on multiple subsequent substitutions, which is less elegant.

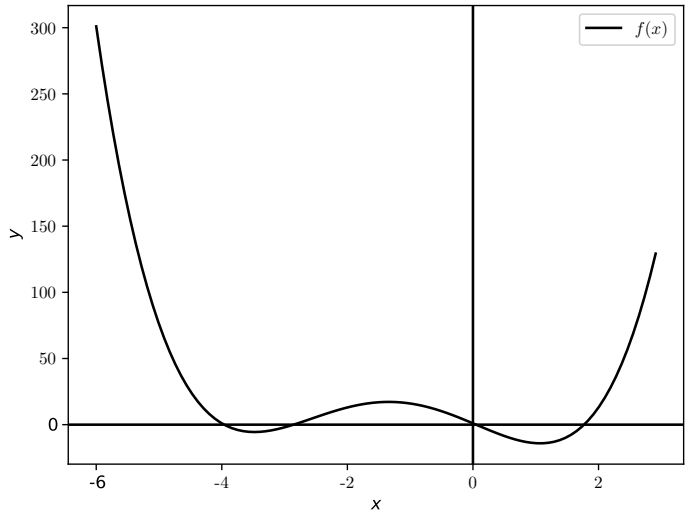

Figure 2.7: The quartic equation $f(x) = x^4 + 5x^3 - x^2 - 20x + 1$, with four real roots.

To factor the quartic in Equation 2.17, it must be transformed into the form

$$(P+Q)(P-Q) = P^2 - Q^2 \stackrel{!}{=} x^4 + ax^3 + bx^2 + cx + d, \tag{2.18}$$

where $P \pm Q$ are quadratic equations. In the first step, P and Q are chosen as

$$P = x^2 + \frac{a}{2}x - \frac{y}{2}, \tag{2.19}$$

$$Q = ex + f, \tag{2.20}$$

where e, f and y are yet to be determined. From Equation 2.18 it must hold that

$$\left(x^2 + \frac{a}{2}x - \frac{y}{2}\right)^2 - (ex+f)^2 \stackrel{!}{=} x^4 + ax^3 + bx^2 + cx + d. \tag{2.21}$$

This is different to the factorization approach given by Turnbull [Tur57], however Equation 2.21 will lead to a simpler cubic resolvent. Expanding the left hand side, one arrives at

$$x^4 + ax^3 + \left(\frac{a^2}{4} - y - e^2\right)x^2 + \left(-\frac{ay}{2} - 2ef\right)x + \left(-f^2 + \frac{y^2}{4}\right). \tag{2.22}$$

By comparing coefficients with the right hand side of Equation 2.21, one obtains the equations

$$b = \frac{a^2}{4} - y - e^2, \qquad c = -\frac{ay}{2} - 2ef, \qquad d = -f^2 + \frac{y^2}{4}. \tag{2.23}$$

From these the unknowns e, f and ef are formulated:

$$e^2 = \frac{a^2}{4} - b - y, \qquad ef = -\frac{ay}{4} - \frac{c}{2}, \qquad f^2 = \frac{y^2}{4} - d. \tag{2.24}$$

In order for $(P+Q)(P-Q)$ to hold, y must be chosen. From e^2, f^2, and ef an equation for y is constructed as there are no mutual dependencies between the equations with

$$(ef)^2 = e^2 f^2 \tag{2.25}$$

$$\iff \left(-\frac{ay}{4} - \frac{c}{2}\right)^2 = \left(\frac{a^2}{4} - b - y\right)\left(\frac{y^2}{4} - d\right) \tag{2.26}$$

$$\iff \frac{a^2}{4}y^2 + acy + c^2 = \frac{a^2}{4}y^2 - a^2 d - by^2 + 4bd - y^3 + 4dy. \tag{2.27}$$

Rearranging everything to one side yields

$$y^3 + by^2 + (ac - 4d)y + (c^2 + a^2 d - 4bd) = 0, \tag{2.28}$$

which is the cubic resolvent. Solving for y yields up to three real roots. Only one of these roots is required to construct the quadratic equations $(P \pm Q)$, whose solutions correspond to the roots of the quartic.

The quadratic equations then read [Her95]

$$0 = x^2 + Gx + H, \tag{2.29}$$

$$0 = x^2 + gx + h, \tag{2.30}$$

with

$$G = \frac{a}{2} + e, \tag{2.31}$$

$$H = -\frac{y_1}{2} + f, \tag{2.32}$$

$$g = \frac{a}{2} - e, \tag{2.33}$$

$$h = -\frac{y_1}{2} - f, \tag{2.34}$$

which can be solved immediately.

Solving the Cubic Resolvent

The general solution of the cubic equation was derived by Tartaglia, however today it is associated with Cardan [Tur57]. Again, the derivation below follows the approach given by Turnbull [Tur57] which is slightly modified to obtain the equations given by Herbison-Evans [Her95]. Consider the general cubic equation

$$x^3 + px^2 + qx + r = 0, \tag{2.35}$$

which is transformed by the substitution $x = y - \frac{p}{3}$ to

$$y^3 + \underbrace{\left(q - \frac{p^2}{3}\right)}_{u} y + \underbrace{\frac{2p^3}{27} - \frac{pq}{3} + r}_{v} = 0. \tag{2.36}$$

Assuming $y = z + n$ and inserting in Equation 2.36 yields

$$y^3 = z^3 + n^3 + 3zny. \tag{2.37}$$

Slight rearranging and coefficient comparison with Equation 2.36 yields

$$0 = y^3 + \underbrace{(-3zn)}_{\equiv u} y + \underbrace{(-z^3 - n^3)}_{\equiv v},$$

as this is again the transformed cubic. Therefore the equations

$$-u = 3zn, \tag{2.38}$$
$$-v = z^3 + n^3 \tag{2.39}$$

hold as well.

Next a quadratic equation whose two roots $\lambda_{1,2}$ are exactly z^3 and n^3 such that Equation 2.39 and Equation 2.38 hold simultaneously is constructed. This requires the usage of Vieta's formulas [Wei18b], which state that for the general quadratic equation

$$x^2 + \zeta x + \tau \tag{2.40}$$

with roots $x_{1,2}$ it holds that

$$-\zeta = x_1 + x_2, \qquad \tau = x_1 x_2. \tag{2.41}$$

Equation 2.39 gives the condition that the sum of the roots shall be $-v$. Similarly Equation 2.38 requires that the product of the roots shall be $\frac{-u^3}{27}$, yielding the quadratic

$$\lambda^2 + v\lambda - \frac{u^3}{27} = 0. \tag{2.42}$$

Finally the solutions to

$$y = z + n = \sqrt[3]{\frac{-v}{2} + \sqrt{\frac{v^2}{4} + \frac{u^3}{27}}} + \sqrt[3]{\frac{-v}{2} - \sqrt{\frac{v^2}{4} + \frac{u^3}{27}}} \tag{2.43}$$

contains all roots since

$$\lambda_1 = \frac{-v^2}{2} + \sqrt{\frac{v^2}{4} + \frac{u^3}{27}} = z^3, \tag{2.44}$$
$$\lambda_2 = \frac{-v^2}{2} - \sqrt{\frac{v^2}{4} + \frac{u^3}{27}} = n^3 \tag{2.45}$$

are the solutions to the quadratic equation. Note that in Equation 2.43 for each cubic root three solutions can be obtained for z and n. Therefore there are 3×3 combinations of solutions $y = z + n$. However, not all combination are valid. The combinations are examined by defining $\omega = \frac{1}{2} + i\frac{\sqrt{3}}{2}$. Then $\omega^3 = 1$, $\omega^6 = 1$ and therefore

$$z^3 = (1z)^3 \text{ or } (\omega z)^3 \text{ or } (\omega^2 z)^3, \tag{2.46}$$
$$n^3 = (1n)^3 \text{ or } (\omega n)^3 \text{ or } (\omega^2 n)^3 \tag{2.47}$$

hold as well. Equation 2.39 always holds for all combinations of z and n, which is not the case for Equation 2.38. The three combinations

$$y_1 = z + n, \tag{2.48}$$
$$y_2 = \omega z + \omega^2 n, \tag{2.49}$$
$$y_3 = \omega^2 z + \omega n \tag{2.50}$$

fulfill both equations, which therefore describe the three roots.

The discriminant of the cubic is

$$\triangle = \frac{v^2}{4} + \frac{u^3}{27}. \tag{2.51}$$

This expression appears in the solution in Equation 2.43. If $\triangle > 0$, then the terms $\sqrt{\triangle}$ of Equation 2.43 are real and taking the real cubic roots to obtain $y_1 = z + n$ gives one real root. Since the Equations for y_2 and y_3 hold, both other roots are necessarily complex.

Equation 2.44 and Equation 2.45 are rearranged in the following way:

$$\begin{aligned} z^3, n^3 &= \frac{-v^2}{2} \pm \sqrt{\frac{v^2}{4} + \frac{u^3}{27}} \\ &= \frac{-v^2}{2} \pm \frac{1}{2} \underbrace{\sqrt{v^2 + 4\frac{u^3}{27}}}_{w} \\ &= \frac{-v \pm w}{2}. \end{aligned}$$

Equation 2.39 now yields

$$n \equiv -\frac{u}{3\sqrt[3]{\frac{w-v}{2}}} = -\frac{u}{3}\sqrt[3]{\frac{2}{w-v}}. \tag{2.52}$$

Now with $y_1 = z + n$ it follows that

$$y_1 = \sqrt[3]{\frac{w-v}{2}} - \frac{u}{3}\sqrt[3]{\frac{2}{w-v}}, \tag{2.53}$$

which is the solution for one real root given by Herbison-Evans [Her95]. Finally, to obtain the actual root the substitution done in Equation 2.36 has to be reversed, i.e.

$$x_1 = y_1 - \frac{p}{3}. \tag{2.54}$$

If $\triangle < 0$ holds, then the solution of $\sqrt{\triangle}$ will be purely complex. The solutions before taking the cubic root are therefore

$$z^3, n^3 = \underbrace{\frac{-v}{2}}_{\alpha} \pm i \underbrace{\sqrt{-\triangle}}_{\beta}. \tag{2.55}$$

Written in polar form, this is equivalent to

$$z^3, n^3 = \underbrace{r\cos\theta}_{\equiv\alpha} \pm i \underbrace{r\sin\theta}_{\equiv\beta} \tag{2.56}$$

where due to the nature of the polar form, in the complex plane the length r is

$$\begin{aligned} r &= \sqrt{\alpha^2 + \beta^2} \\ &= \sqrt{\frac{-u^3}{27}} = \sqrt{-\left(\frac{u}{3}\right)^3}, \end{aligned}$$

and the angle θ is

$$\cos\theta = \frac{\alpha}{r} = \frac{-v}{2r} \implies \theta = \cos^{-1}\frac{-v}{2r}.$$

Since $y = z + n$ holds the roots are also describable as

$$\begin{aligned} y &= \sqrt[3]{r\cos\theta + ri\sin\theta} + \sqrt[3]{r\cos\theta - ri\sin\theta} \\ &= \sqrt[3]{r}\left(\sqrt[3]{\cos\theta + i\sin\theta} + \sqrt[3]{\cos\theta - i\sin\theta}\right) \\ y_{m+1} &= \sqrt[3]{r}\left(\cos\frac{\theta + 2m\pi}{3} + i\sin\frac{\theta + 2m\pi}{3} + \cos\frac{\theta + 2m\pi}{3} - i\sin\frac{\theta + 2m\pi}{3}\right) \\ &= 2\sqrt[3]{r}\cos\frac{\theta + 2m\pi}{3}, \end{aligned}$$

for $m \in \{0, 1, 2\}$. The switch to y_{m+1} is the result of taking the cubic root of a complex number $p = \rho(\cos\phi + i\sin\phi)$ in polar form [MS05] which yields three solutions p_1, p_2 and p_3 with

$$p_{m+1} = \sqrt[3]{\rho}\left(\cos\frac{\phi + 2\pi m}{3} + i\sin\frac{\phi + 2\pi m}{3}\right). \tag{2.57}$$

Therefore all obtainable roots y_m are real. The solutions given by Herbison-Evans are obtained by

$$\begin{aligned} s &= \sqrt[3]{r} = \left(\left(-\frac{u}{3}\right)^3\right)^{\frac{1}{6}} = \sqrt{-\frac{u}{3}}, \\ k &= \frac{\theta}{3} = \frac{\cos^{-1}\frac{-v}{2r}}{3} = \frac{\cos^{-1}\frac{-v}{2s^3}}{3}. \end{aligned}$$

Now s and k are used to describe the roots as

$$y_{m+1} = 2s\cos\left(k + \frac{2m\pi}{3}\right). \tag{2.58}$$

The equality $\cos(\gamma + \delta) = \cos\gamma\cos\delta - \sin\gamma\sin\delta$ is used on this equation's cos term which yields

$$\cos\left(k + \frac{2m\pi}{3}\right) = \cos k\cos\frac{2m\pi}{3} - \sin k\sin\frac{2m\pi}{3}. \tag{2.59}$$

Finally, the three roots are explicitly

$$y_1 = 2s\cos k, \tag{2.60}$$

$$y_2 = s(-\cos k + \sqrt{3}\sin k), \tag{2.61}$$

$$y_3 = s(-\cos k - \sqrt{3}\sin k). \tag{2.62}$$

Again the substitution of Equation 2.36 is reversed, yielding

$$x_1 = y_1 - \frac{p}{3}, \qquad x_2 = y_2 - \frac{p}{3}, \qquad x_3 = y_3 - \frac{p}{3}. \tag{2.63}$$

Numerical Issues

Even though the Ferrari approach solves the quartic equation completely analytically, floating point precision is still an issue to consider. Floating point operations may overflow or introduce heavy roundoff errors [Her95]. In case of the ray-torus intersection, this will lead to visual artifacts such as holes in the surface, if intersections are missed. Herbison-Evans [Her95] numerically stabilized the Ferrari and the Cardan approach to reduce numerical errors, which is described in the following.

When a particular quartic is examined, depending on its coefficient signs and the sign of the cubic resolvent's root, the coefficients of the factored quadratic equations may be numerically stable or erroneous. This occurs due to the nature of the calculations that have to be computed. Equation 2.24 requires that sums of terms have to be calculated. For example, to compute e^2, b and y have to be positive in order for the calculation to be stable. Otherwise, they might cancel to a small value $|e^2|$. In the face of numerical inaccuracy, the sign of e^2 may then become negative. The same argument holds for the computation of f^2 and ef. However, if at least two of the equations are stable and one of them is ef, then the originally unstable value is obtainable by rearranging ef.

Equations 2.23 allows stabilization of g (used in Equation 2.30) by rearrangement as

$$\begin{aligned} & b + y = \frac{a^2}{4} - e^2 \\ \Longleftrightarrow\ & b + y = \left(\frac{a}{2} + e\right)\left(\frac{a}{2} - e\right) \\ \Longleftrightarrow\ & \frac{b+y}{\frac{a}{2}+e} = \frac{a}{2} - e \\ \Longleftrightarrow\ & \frac{b+y}{G} = g, \end{aligned}$$

which gives stable g in the face of $\text{sign}(a) = \text{sign}(e)$ and $\text{sign}(b) = \text{sign}(y)$. One may rearrange this to

$$G = \frac{b+y}{g} \tag{2.64}$$

to obtain stable G when $\text{sign}(a) \neq \text{sign}(e)$. Similarly Equations 2.23 yield

$$H = d/h, \qquad h = d/H \tag{2.65}$$

for stable H if $\text{sign}(y) = \text{sign}(f)$, or stable h if $\text{sign}(y) \neq \text{sign}(f)$. For the solution of the cubic equation, Equation 2.53 can be stabilized as well. The term $\frac{w-v}{2}$ may become negative due to numerical inaccuracy if $v > 0$. However the term may by expanded in the nominator and denominator with $w + v$ which yields the alternative formulation

$$y = \sqrt[3]{\frac{w+v}{2}} - \frac{u}{3}\sqrt[3]{\frac{2}{w+v}}, \tag{2.66}$$

which is then stable. Since $x_1 \leq x_2 \leq x_3$, the third root or first root is most useful for computation, as it can be used to keep values from becoming very big in terms of absolute value or even overflowing.

2.2.2 Iterative Solvers

Iterative solvers for the root finding problem are useful if exact roots are not required. In the ray-torus intersection problem, if the error to the true roots is kept small enough, it will not appear visible in computed images. A general approach that is also applicable to arbitrary polynomials consists of computing the Eigenvalues of the *companion matrix* of a quartic which is constructed as

$$A = \begin{pmatrix} -a_0 & -a_1 & -a_2 & -a_3 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{2.67}$$

The Eigenvalues are equivalent to the quartic's roots, since A's characteristic polynomial is equivalent to the quartic [PTVF92]. However this requires the usage of expensive Eigenvalue iteration algorithms.

Newton-Raphson Method

Another popular approach is based on the Taylor-Expansion of a generic continuous function $f(x)$ yielding

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{1}{2}f''(x)\delta^2 + ..., \tag{2.68}$$

which is called the *Newton-Raphson method* [PTVF92]. At the current position x_t the step length

$$\delta = -\frac{f(x_t)}{f'(x_t)} \tag{2.69}$$

is evaluated and used to update the current x_t with

$$x_{t+1} \leftarrow x_t + \delta \tag{2.70}$$

until convergence to a root [PTVF92]. While this method converges quadratically with each step when started close to a root, it may diverge if the initial x_0 is chosen arbitrarily. Another disadvantage is that convergence of the method only implies that a single root was found. It is possible to deflate the polynomial with each found root, however this may be numerically unstable.

Bairstow's Method

Bairstows Method computes a factorization of an arbitrary polynomial $f(x)$ of degree n of one quadratic term and another polynomial $q(x)$ of degree $n-2$. The following derivation follows Press et al. [PTVF92]. Let

$$x^2 - 2\alpha x + \left(\alpha^2 + \beta^2\right) \equiv x^2 + Bx + C \tag{2.71}$$

be the quadratic with solutions $x_{1,2} = \alpha \pm i\beta$.

Now this quadratic is used to represent $f(x)$ as

$$f(x) = (x^2 + Bx + C)q(x) + r(B,C)x + s(B,C) \tag{2.72}$$

where $q(x)$ is a quadratic and $r(B,C)x + s(B,C)$ is the resulting rest term. Note the rest term's dependency on the coefficients B and C. The unknowns $q(x)$, $r(B,C)$ and $s(B,C)$ can be obtained by polynomial division. B and C divide $f(x)$ without remainder if $r(B,C), s(B,C) = 0$. The method now seeks to find coefficients B and C such that they correspond to roots of r and s. In case of the quartic after B and C were determined, immediately $q(x)$ and $(x^2 + Bx + C)$ can be solved to obtain all roots.

Similar to the Newton-Raphson method the Taylor expansion can be used to obtain step sizes δB and δC for both r and s simultaneously as

$$r(B + \delta B, C + \delta C) \approx r(B,C) + \frac{\partial r}{\partial B}\delta B + \frac{\partial r}{\delta C}\delta C, \tag{2.73}$$

$$s(B + \delta B, C + \delta C) \approx s(B,C) + \frac{\partial s}{\partial B}\delta B + \frac{\partial s}{\partial C}\delta C, \tag{2.74}$$

which results in the linear system of equations

$$-r(B,C) = \frac{\partial r(B,C)}{\partial B}\delta B + \frac{\partial r(B,C)}{\delta C}\delta C, \tag{2.75}$$

$$-s(B,C) = \frac{\partial s(B,C)}{\partial B}\delta B + \frac{\partial s(B,C)}{\partial C}\delta C, \tag{2.76}$$

by setting the left hand side of both equations to zero. The unknowns $\frac{\partial r(B,C)}{\partial B}$, $\frac{\partial r(B,C)}{\delta C}$, $\frac{\partial s(B,C)}{\partial B}$, $\frac{\partial s(B,C)}{\partial C}$ as well as $r(B,C)$ and $s(B,C)$ are to be determined in each iteration step. Note that here B, C are seen as fixed in the current iteration step to compute δB and δC.

Solving this linear system yields

$$\delta B = \frac{\frac{\partial s(B,C)}{\partial C}r(B,C) - \frac{\partial r(B,C)}{\partial C}s(B,C)}{\gamma} \tag{2.77}$$

$$\delta C = \frac{\frac{\partial r(B,C)}{\partial B}s(B,C) - \frac{\partial s(B,C)}{\partial B}r(B,C)}{\gamma} \tag{2.78}$$

where

$$\gamma = \frac{\partial r(B,C)}{\partial C}\frac{\partial s(B,C)}{\partial B} - \frac{\partial s(B,C)}{\partial C}\frac{\partial r(B,C)}{\partial B}. \tag{2.79}$$

Since δB and δC are step sizes towards the roots of r and s, the iteration is applied as

$$B_{t+1} = B_t + \delta B, \qquad C_{t+1} = C_t + \delta C. \tag{2.80}$$

The partial derivatives are obtained by computing the derivate of Equation 2.72 with respect to C and separately for B on both sides, resulting in

$$\frac{\partial f(x)}{\partial B} = \frac{\partial\left((x^2 + Bx + C)q(x) + r(B,C)x + s(B,C)\right)}{\partial B} \tag{2.81}$$

$$\iff 0 = (x^2 + Bx + C)\frac{\partial q(x)}{\partial B} + q(x) + \frac{\partial r(B,C)}{\partial B} + \frac{\partial s(B,C)}{\partial B}, \tag{2.82}$$

and

$$\frac{\partial f(x)}{\partial B} = \frac{\partial \left((x^2 + Bx + C)q(x) + r(B,C)x + s(B,C)\right)}{\partial B} \tag{2.83}$$

$$\iff 0 = (x^2 + Bx + C)\frac{\partial q(x)}{\partial C} + q(x)x + \frac{\partial r(B,C)}{\partial C} + \frac{\partial s(B,C)}{\partial C}. \tag{2.84}$$

Dividing $q(x)$ in Equation 2.72 by $x^2 + Bx + C$ yields

$$q(x) = (x^2 + Bx + C)g(x) + \hat{r}(B,C)x + \hat{s}(B,C) \tag{2.85}$$

where $g(x)$ is now of degree $n-4$. Again, $g(x)$, $\hat{r}(B,C)$ and $\hat{s}(B,C)$ can be obtained by polynomial division. Rearranging Equation 2.82 by pulling $q(x)$ on the other side and comparing it with Equation 2.85 results in

$$\frac{\partial r(B,C)}{\partial C} = -\hat{r}(B,C), \qquad \frac{\partial s(B,C)}{\partial C} = -\hat{s}(B,C). \tag{2.86}$$

When the roots $x_{1,2}$ are put into Equation 2.85, it holds that

$$q(x_1) = \hat{r}(B,C)x_1 + \hat{s}(B,C), \tag{2.87}$$

$$q(x_2) = \hat{r}(B,C)x_2 + \hat{s}(B,C), \tag{2.88}$$

as the quadratic multiplied with $g(x)$ will vanish. Putting roots $x_{1,2}$ into Equation 2.82 and replacing $q(x_1)$ and $q(x_2)$ with Equation 2.87 and Equation 2.88 gives

$$\frac{\partial r(B,C)}{\partial B}x_1 + \frac{\partial s(B,C)}{\partial B} = -x_1(\hat{r}(B,C)x_1 + \hat{s}(B,C)) \tag{2.89}$$

$$\frac{\partial r(B,C)}{\partial B}x_2 + \frac{\partial s(B,C)}{\partial B} = -x_2(\hat{r}(B,C)x_2 + \hat{s}(B,C)). \tag{2.90}$$

Vieta's formulas (see Equation 2.41) are now used on the roots of the quadratic to obtain

$$x_1 + x_2 = -B, \qquad x_1 x_2 = C, \tag{2.91}$$

which allow solving of Equation 2.89 and Equation 2.90 yielding

$$\frac{\partial r(B,C)}{\partial B} = B\hat{r} - \hat{s}, \qquad \frac{\partial s(B,C)}{\partial B} = C\hat{r}(B,C). \tag{2.92}$$

Now the iteration according to Equations 2.80 can be computed.

2.3 Acceleration Structures

Ray tracing performance greatly benefits from the usage of spatial data structures, which allow sublinear intersection computation. All these data structures have in common that they either divide the scene's space or divide the scene objects which allows a query ray to exclude most scene objects from intersection tests. The simplest structure is the *uniform spatial subdivison* which just divides the space into a grid with uniform side lengths per dimension [SM09]. Intersection requires the ray to check all the objects associated

with every grid cell it passes through for intersection in an incremental fashion, until an intersection is found. The advantage of this method is simple implementation and the possibility of rapid construction of the grid which allows interactive ray tracing, as shown by Wald et al. [WIK+06] and Gribble et al. [GIK+07]. Grids are also used to accelerate neighborhood computation in molecular datasets for the same reasons [KBE09; LBPH10].

Bounding Volume Hierarchies (BVHs) [PH10] or *hierarchical bounding boxes* [SM09] are a tree based approach that associates objects with tree leafs. In general, each object has a bounding box. Each parent node now has the union of all child node bounding boxes as its own bounding box, up until the root is reached, which has the bounding box of the entire scene. Traversal begins at the root and recursively traverses the tree's nodes. A valid intersection requires that the node's bounding box is intersected first, otherwise its entire subtree can be skipped. In case of a bounding box intersection the node's children have to be recursively examined further. Since such a hierarchy does allow overlapping bounding boxes of child nodes, all returned intersection have to be checked for the closest intersection.

Binary Space Partitioning [SM09] trees (BSP trees) are a approach that combines space division with a hierarchical structure. Tree nodes contain cutting planes that divide the space in two halfspaces, e.g. the root plane splits the entire scene in half. Depending on the ray origin and direction, halfspaces can be skipped entirely, e.g. if the ray is inside a halfspace and points away from the plane, it will never intersect any object in the other halfspace.

Naturally, the tree based approaches are more expensive to construct than grids, depending on the used heuristics. Grids also have the disadvantage that the same object may be tested more than once for intersection, if the object overlaps multiple cells at once [PH10]. With many scene objects ray tracing becomes more efficient than object order rendering methods such as (software) rasterization [SM09; WJA+17], as occluded objects in dense scenes are skipped, while rasterization based approaches will then produce overdraw. See Figure 2.8 for examples of all three structures.

2.4 Software

This section contains an introduction to the software that was used to implement this work. First, a vectorized C-variant is introduced that offers performance benefits for parallelizeable problems computed on CPUs. Both the ray tracing acceleration library Embree and OSPRay, which makes use of Embree are described briefly. The geometry described in this work was implemented in OSPRay. Lastly, the MegaMol framework is described which was used to integrate the implemented geometry.

2.4.1 ISPC

Modern CPUs have the ability to execute the same operation in a single step for multiple data values in SIMD (single instruction, multiple data) vector units. This offers the possibility for low level parallel code execution in addition to threaded execution over

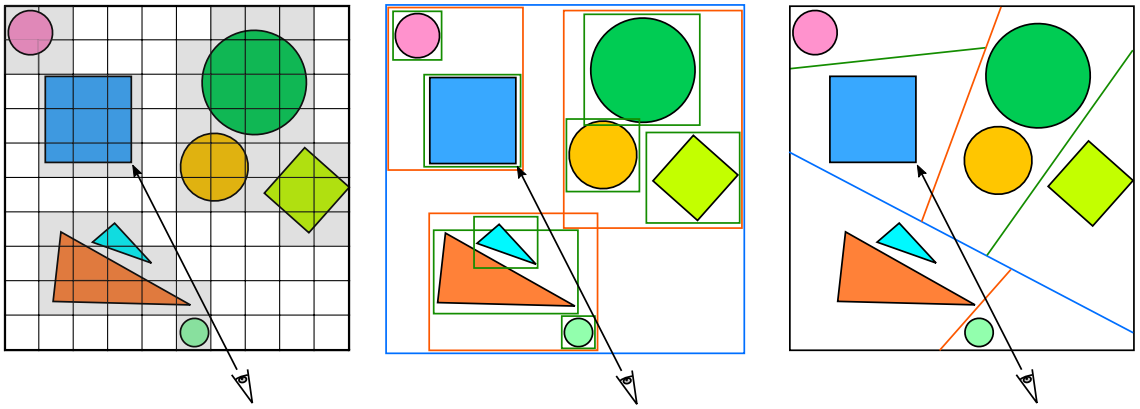

Figure 2.8: From left to right: Grid, bounding volume hierarchy and binary space partitioning. The grid avoids intersection with the geometry on the right. However, it can be seen that sometimes the same geometry is checked multiple times for intersection. In the bounding volume hierarchy, all bounding boxes on the same tree level are colored the same, similar to the planes in the binary space partitioning example. The bounding volume hierarchy allows skipping of the top-right subtree. Binary space partitioning is less effective in this example as only the pink sphere in the top-left can be skipped after the intersection with the blue rectangle was found.

multiple cores. While current compilers like gcc[1] may support low level capabilities for writing vectorized code, this remains a highly challenging task for programmers, especially if portable code for complex algorithms is required. The *Intel SPMD Program Compiler*[2](ISPC) aims to solve this problem by offering a C-like language that compiles to ordinary object files that can be linked together with C/C++ code, by compiling high level code to SSE or AVX vector instructions [PM12]. SMPD stands for single-program, multiple-data, which is the programming model used by ISPC. Functions written in ISPC are exposed with the `export` keyword. On the C/C++ side, a specialized header file is included that declares the function so it can be called from C/C++ code like any other function. Function arguments such as pointers and references allow direct access to client memory on the ISPC side without copying any data, in contrast to GPU programming APIs. The authors report a speedup of roughly three to eight times depending on the problem compared to scalar C code, when executed on a single core with vector width eight.

Execution Model

Conceptually, upon function entry a *gang* of *program instances* is started, where the number of gang members n is generally up to twice the SIMD vector width [PM12]. Execution however differs from scalar code. Each program variable is defined as `varying` or `uniform`,

[1] https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html
[2] https://ispc.github.io/index.html

Listing 2.1 ISPC example code that computes a discretized representation of the function $z = \sin x + \sin y$ in the domain $[0, \pi) \times [0, \pi) \subset \mathbb{R}^2$ vectorized over consecutive entries in of the result array.

```
1  export void added_sines(uniform int numX, uniform int numY, uniform float *uniform result) {
2
3    for (uniform int y = 0; y < numY; y++) {
4      uniform const float yCoord = PI*((float)y)/numY;
5      foreach(x = 0 ... numX) {
6        varying const float xCoord = PI*((float)x)/numX;
7        varying int index = x + numX*y;
8        varying float z = sin(xCoord) + sin(yCoord);
9        result[index] = z;
10     }
11   }
12 }
```

where `varying` variables contain n values each, while `uniform` variables are invariant among all program instances. Therefore computations that are intended to run in parallel should use variables with the `varying` keyword. During the execution, if a operation (such as $+$) is encountered that is written to a `varying` variable, the operation is executed in the CPU's SIMD unit. Control flow statements such as `if` involving `varying` variables may lead to divergence of executing program instances. All instances always execute all encountered program statements, however a *execution mask* determines the current active program instances. If a conditional statement leads to diverging execution, the execution mask is used such that only active program instance's operations can have side effects. Computation involving purely `uniform` variables can be compared to scalar execution. While a `varying` variable can be written with a `uniform` variable, the other way around is explicitly forbidden as it would be ambiguous.

Performance

Since conditional statements cannot be skipped if at least one program instances wants to execute the inside body, all other program instance have to execute in parallel as well (and compute garbage). In general due to execution path divergence branching implies a performance penalty. Another issue is the memory layout of computation data. The often used *array of structs* (AoS) layout can be disadvantageous, as load and store operations access data sequentially. On the other hand, the *struct of arrays* (SoA) layout allows vectorized loading of coherent data values, e.g. the x-variable of n position vectors can be loaded in one step if they are located coherently in one array.

2.4.2 Embree

The *Embree*[3] framework is a lightweight CPU ray tracing library that offers efficient implementation of bounding volume hierarchies that allow vectorized ray queries via its API [WWB+14]. Since version 2.0, the ISPC language is used for the implementation of ray tracing kernels and can be used for function calls to the API. The BVH relies on axis aligned bounding boxes. Both single ray and ray packet traversal is supported. Depending on the architecture, single ray traversal is either vectorized over the BVH nodes or the components of the intersection variables (e.g. intersection point). This is well suited for incoherent rays, such as the ray queries required for sampling a diffuse reflection. In contrast, ray packet tracing intersects a set of coherent rays with a single BVH node in each step. The OSPRay framework currently only uses this approach, even though Embree is able to use both methods simultaneously on the same BVH. In addition to triangles, users are able to implement custom geometries by implementing call back function called on intersection tests and bounding box queries.

2.4.3 OSPRay

OSPRay[4] [WJA+17] is a rendering framework that uses CPU ray tracing to aid implementation of scientific visualization methods. The framework is build on top of the Embree API and uses ISPC for performance critical code such as intersection computations. Currently, OSPRay abstracts away commonly used entities such as renderers, cameras, materials, lights and geometries which are not provided by Embree's API by default. Additionally, support for distributed rendering is offered. These entities are offered through a string based API, which allows users to write custom extension. In the high performance computing context (HPC), the usage of a CPU based rendering solutions is justified by the fact that distributed compute nodes oftentimes do not contain a dedicated GPU, which forces software rasterization implementations. Rasterization also complicates implementation of advanced shading effects that are oftentimes beneficial for visualization purposes [GP06]. Another advantage of CPU ray tracing is the access to the entire available memory of compute nodes, which oftentimes exceeds GPU memory, if available. However, full access is required for techniques such as rendering of large volume datasets [WJA+17].

2.4.4 MegaMol

MegaMol[5] [GKM+15] is a framework intended to simplify development of visualization techniques for particle data sets, with a focus on GPU based implementations. Developers are aided by commonly used functionality such as shader management and UI interaction, the *core* of the framework. *Plugins* augment the core by implementing a collection of *modules* which are abstract classes that each already offer some generic functionality. Modules may communicate with other modules. For example, a rendering module may

[3] https://embree.github.io/
[4] https://www.ospray.org/
[5] https://megamol.org/

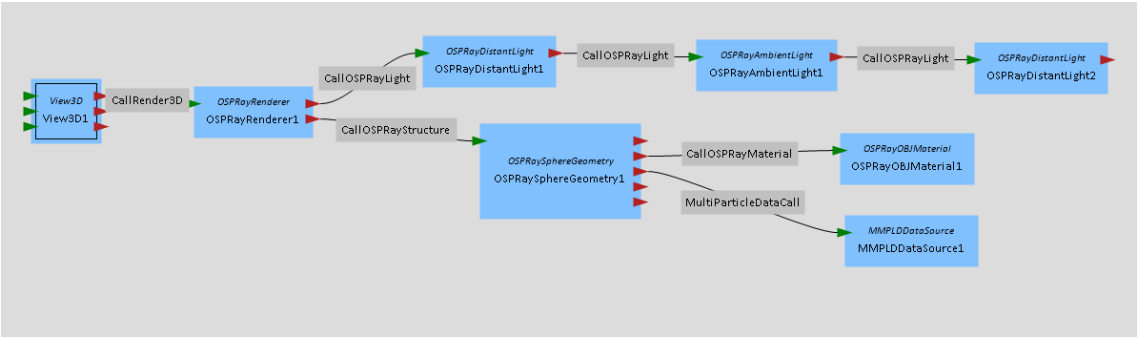

Figure 2.9: Example of a typical MegaMol graph, created with MegaMol's configurator tool. Here the modules of the OSPRay integration are shown. The View3D module controls general parameters such as camera position and orientation. The OSPRayRenderer module wraps an OSPRay renderer (''scivis'' or ''pathtracer''). Both OSPRayAmbientLight and OSPRayDistantLight control scene illumination. The OSPRaySphereGeometry module wraps OSPRay's sphere geometry. MMPLDDataSource is used to load the data into the sphere geometry. Finally, the OSPRayOBJMaterial module defines the geometry's material.

ask a data loading module for data, such as particle datasets. If such communication is required, this is modeled as a *call*, which connects both modules. Calls are generic and allow connection of modules with matching interfaces by offering callback functions. For example commonly particle datasets consist of particles positions, colors and radii. There may be different loaders for different files format, but they all allow connection with the same call, abstracting away the specific implementation. This modular nature allows reusage of modules together with newly developed ones. Modules and calls can be seen as nodes and edges of a graph structure.

OSPRay is largely integrated in MegaMol as individual modules, such as the various geometries and material types [RKRE17], allowing rendering of datasets with CPU ray tracing in contrast to the GPU based approaches of the rest of the framework. Figure 2.9 shows a MegaMol module graph that uses the OSPRay integration.

3 Molecular Dynamics Visualization

The visualization subfield *molecular dynamics visualization* attempts to aid domain experts in the understanding of structure, behavior and mutual interaction of molecules of various kinds. A molecule consists of a number of atoms, which are held together by attracting forces, for example produced by chemical bonds between the atoms. Molecules concerning a living system are called biomolecules [KKL+15], for example proteins. Visualization aims to couple the molecule's shape with additional information such as electrical and thermal data, or provide abstraction of the molecule's properties by mapping secondary structures to geometric primitives. *Molecular surfaces* serve as a means to convey the *accessibility*, i.e. the potential for interaction between molecules and other, independent entities. Surfaces allow detection and visualization of spatial characteristics such as cavities (clefts, tunnels, pores and others) shaped by the atoms of a molecule [KKL+16]. Interesting interactions between molecules oftentimes occurs in these positions. Relevant datasets are either produced by simulations, or results of X-Ray crystallography [KKL+15].

3.1 Simulation of Molecules

Adcock and McCammon [AM06] gave an extensive overview over modern molecular simulation techniques. One classic approach known as *molecular dynamics* (MD) simulation models molecules as point masses connected by mechanical springs [AM06]. Hereby a spring between atoms models bounding forces. For example, *Hooke's law* is a potential candidate to model attraction and repulsion between bonded atoms, while the *Lennard-Jones potential* models interaction of non bounded atoms. For each state of the molecule, a force field function describes its current potential energy, which consists of a sum of terms parameterized by the molecule's atom positions in space. The model's parameterization is obtained from experimentation or theoretical considerations of quantum mechanics. These models disregard computation of quantum effects themselves, as these are currently unfeasible to simulate [AM06; KKL+15] in large systems for extended periods of time. Bonds between atoms are predefined, which does not allow simulation of reactions at binding sites. Combined *quantum mechanical/molecular mechanical* methods allow simulation of such interactions, however the modeling of quantum mechanical effects increases computational costs [ST09]. In the simulation, iteratively at each time step the applied force of each atom position is computed from the current potential energy and applied to solve an equation of motion numerically. The potential energy's derivative with respect to the position gives the required force vector [AM06; Jen06]. The molecule may be simulated in vacuum or submerged in water. Such a solvent can be modeled explicitly or implicitly. An explicit model is computationally more expensive as solvent molecules interact with the studied one. Implicit models are more efficient to compute, but may lack the modeling of certain effects

and thus may be less accurate. The result of such simulations are trajectories of atom centers [KKL+15], which allow domain experts to inspect interactions in high temporal resolution for a short duration on nanosecond to microsecond scale. If computational efficiency is of concern, the *Metropolis monte carlo* approach can be used [AM06]. This method is more efficient as it avoids numerical integration to compute the atom positions for each time step. In this method, the molecule's current state is repeatedly perpetuated randomly to obtain new states [AM06; FS01]. If this change results in lowered potential energy of the entire system, it is accepted. Otherwise, the state is only taken with a certain probability. In contrast to the classical approach, there is no relation to time.

3.2 Visualization Techniques

Molecular datasets can be visualized differently according to the information that is to be conveyed. Kozlikova et al. [KKL+15] grouped visualization techniques into two categories: Atomistic and abstract models. Atomistic models show the molecule's atoms directly in some form. The simplest visualization technique consists of rendering the atoms depicted as spheres with the radius chosen as the *van der Waals radius* (see Section 3.2.1). Additionally, this representation can be enhanced to show atomic bonds between atoms as well, known as the *ball-and-stick* visualization [KKL+15]. Here, bonds between atoms are depicted by primitives such as cylinder or lines. Molecular surface representations are another example of atomistic models. In this case, a smooth surface is arranged around the molecule's atoms in such a way that it conveys accessibly of the molecule outer and inner shell for other entities. Section 3.2.1 presents an in-depth description of these representations. Figure 3.1 shows example visualizations of a protein.

On the other hand, abstract models are less concerned with individual atoms. The simplest kind of abstraction consists of reducing several neighboring atoms to one sphere [KKL+15; LPSV14]. Another abstract model is the *cartoon* representation. This technique aims to visualize secondary structures derived from the atom data, commonly amino acid chains such as α-helices and β-sheets [KBE08]. Such chains of atoms are mapped to visual representations like spiraling ribbons and curved cylinder tubes.

Oftentimes, GPU ray casting is used to renderer the atom spheres [Gum03; RE05]. While triangulation of the spheres can be applied, it is not employed often as the number of required triangles for good visual quality implies a severe performance loss. The ray casting technique consists of rendering proxy geometry on the GPU and then applying ray casting for each of its fragments in the fragment shader. For each fragment a ray is cast and intersected with the sphere. If the sphere is hit, shading is applied. Otherwise, the fragment is discarded. This technique can be applied to other, more complex kinds of primitives as well, such as general quadratics [SWBG06], and allows rendering with interactive framerates. Another advantage in contrast to triangularization is the high visual quality such surfaces offer, independent of viewer distance to the primitive. Approximate global illumination effects such as shadows [KRZ+17; SWBG06], ambient occlusion [GKSE12] and diffuse indirect illumination [SVGR16] are supported while still allowing interactive rendering of up to 10^6 particles. Such physically plausible effects can be used to improve the user's spatial

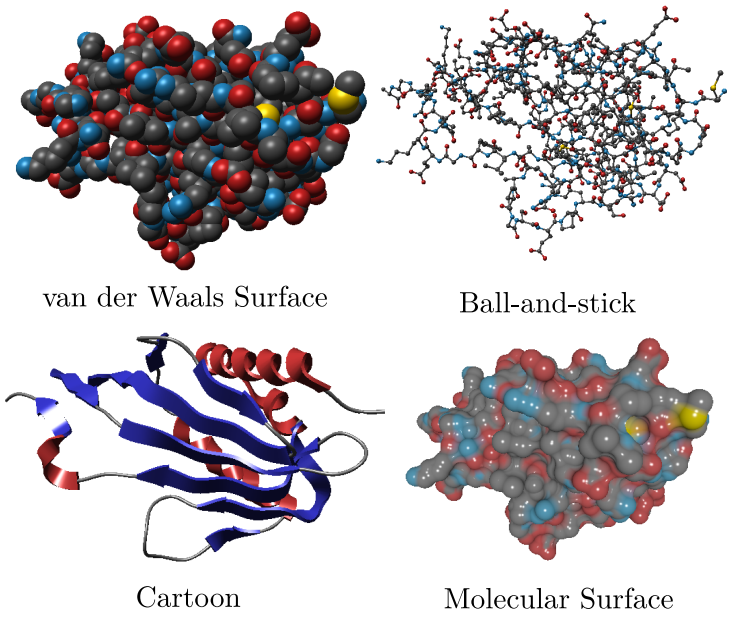

Figure 3.1: Four basic representations for the protein *1ogz*. The van der Waals, ball-and-stick and cartoon renderings were created with MegaMol [GKM+15], the molecular surface was rendered by the implementation described in this work.

impression of the visualized data [GP06]. On the other hand, non realistic techniques can be applied to enhance perception, such as *toon shading* and *halos* to highlight edges, or depth cues such as z-value dependent darking [KKL+15; RE05].

3.2.1 Molecular Surfaces

Let $S = \{\sigma_0, ..., \sigma_{N-1}\}$ be the input dataset of spheres, which represent atoms of a molecule. Each sphere σ_i has a corresponding position p_i and radius R_i. The simplest representation of a molecule is the *van der Waals surface* [KKL+15], which consists of spheres representing the atoms, where each sphere's radius correspond to the van der Waals radius of the corresponding atom. Another simple representation is the *solvent accessible surface* (SAS) [KKL+15; Ric77]. A probe with radius R_{probe} approximates the shape of a solvent molecule by a sphere, e.g. H_2O, which implies a probe radius of $R_{\text{probe}} = 1.4\,\text{Å}$ [Ric77]. Intuitively, one can think of the probe as a solid sphere that is in contact with the van der Waals surface and rolls along the atom spheres. The SAS then describes all possible center point positions the probe is allowed to take while being in contact with the molecule. This surface is obtained by simply extending all atom radii by the *probe radius*.

Solvent Excluded Surface

A more complex representation is the *solvent excluded surface* (SES) [Con83; KKL+15] which was termed *smooth molecular surface* by Richards [Ric77]. Let $N(\sigma_i)$ be the *neighbors* of sphere σ_i [LBPH10], which are positioned close enough together to allow the probe to be in contact with both σ_i and the neighbor, formally

$$N(\sigma_i) = \{\sigma_j | i \neq j \ \wedge \ \|p_i - p_j\| < R_i + R_j + 2R_{\text{probe}}\}. \tag{3.1}$$

Similar to the SAS, a probe rolls over the van der Waals surface. However, not the center point, but surface patches of the probe itself define the surface. The surface is classified in 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 molecule, while the reentrant surface consists of surface patches defined by the probe as it is in contact with two or more atoms spheres. Finally, the union of both the contact and reentrant surface gives the SES [Ric77]. Formally, the SES can be defined as the boundary surface of the interior of the SES I_{SES}[LBPH10], which is the set of all probe positions (even those not in contact with any atom) that do not intersect any atoms, subtracted from $\mathbb{R}^3$, i.e.

$$I_{\text{SES}} = \left\{ \mathbb{R}^3 \setminus \bigcup_{p \in \mathbb{R}^3} \sigma_p | \sigma_p \cap \sigma_i = \emptyset, \ \forall \, i \in \{0, \ ..., \ N-1\} \right\}. \tag{3.2}$$

While the van der Waals surface and the SAS allow discontinuous jumps when spheres are in contact, for the SES all connections between primitives are smooth, i.e. there exists a tangent plane for each contact point [Con83], except at certain singular positions. The relation between all three surface types is shown in Figure 3.2. Algorithms for the computation of the SES are described in Section 3.3.

Ligand Excluded Surface and Molecular Skin Surface

While not of concern in this work, the *ligand excluded surface* [LBH14] and the *molecular skin surface* [CLM08; LBPH10] should be mentioned here as well. The ligand excluded surface does not approximate the solvent as a spherical probe, but considers a ligand molecule's entire shape in different rotation configurations and even deformations. However in turn the corresponding surface can only be approximated with a grid based approach, as there is no analytic solution to this problem yet. The molecular skin surface has the advantage of being C^1 smooth over the entire surface and only being compromised of quadratic primitives, but lacks the physical meaning the SES and the ligand excluded surface provide [KKL+15], as there is no such thing as a probe being in contact with the surface, but a general smoothness parameter that adjusts the surface.

3.2.2 Cavity Visualization

Since molecule cavities are especially interesting due to their interaction potential, techniques for the spatial visualization of molecule cavities were developed [KKL+16]. According to the definitions by Krone et al., cavities hereby include closed off internal cavities, clefts

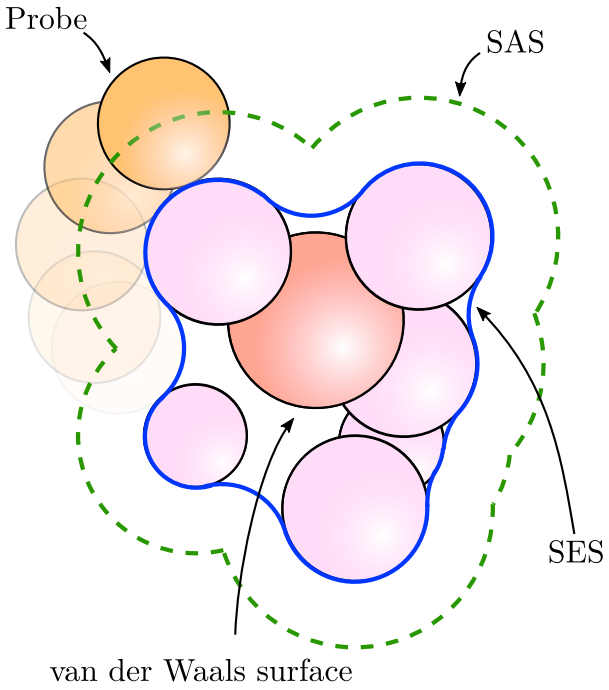

Figure 3.2: Depiction of the most common molecular surfaces. The spheres directly define the van der Waals surface, while the SAS is obtained by extending the spheres by the probe radius, therefore depicting all possible center positions the probe may take. Finally, the SES is defined by the probe positions and their contact points with the spheres. The red sphere is entirely covered and does not contribute to the SES.

accessible from the outside, as well as tunnels that may connect different sides of the molecule. Krone et al. [KKL+16] gave an extensive overview over the many available techniques for cavity extraction and visualization. Grid based techniques for finding cavity compute quantities in a discrete grid and derive cavities from these values. Oftentimes a volume rendering approach is then used to render such solutions to the problem. Spatial subdivision based approaches compute Voronoi-Diagrams from atom positions, and extract cavities from the edges of Voronoi cells. Molecular surfaces can be used to find cavities as well. Jurcik et al. [JPSK16] detected internal cavities by finding isolated components of the solvent excluded surface of molecules. This approach does not extract cavities on the outside of the surface, were interesting binding sites are located. Parulek et al. [PTRV12] used an implicit function $f(\vec{p})$ that models the molecular surface boundary. This function fulfills that at all points on the molecular surface $\vec{p}$ it holds that

$$f(\vec{p}) = 0, \tag{3.3}$$

while for all points $\vec{p}$ inside the space enclosed by the surface,

$$f(\vec{p}) < 0. \tag{3.4}$$

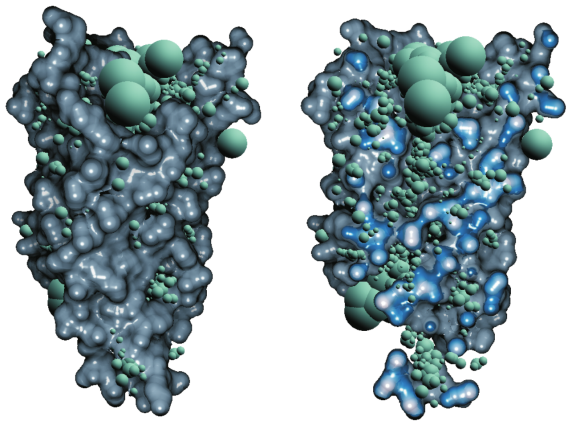

Figure 3.3: Cavity visualizations obtained from the implicit representation of a molecular surface. The image was taken from the paper of Parulek et al. [PTRV12]. Sphere geometry represents detected cavities. In the right image, a clipping plane was additionally applied.

First, the molecule's bounding box is uniformly sampled. Next, the points inside the surface are filtered out by the implicit function. From the remaining points, rays are cast along the gradient direction of $f(\vec{p})$. If the molecular surface is intersected, the points position is adjusted. In the last step, all points that hit the surface closer than some threshold distance are tested for mutual visibility and linked together to multiple graph components representing the cavities, which can then be depicted by spheres and lines. The result of the method is depicted in Figure 3.3. The *ambient occlusion* (AO) technique enables extraction of cavities [Bor11; KRS+13]. Consider a surface point p with normal $\vec{n}_p$, that is illuminated by an environmental light source. Then the ambient occlusion term [Bor11]

$$O_p = \frac{1}{\pi} \int_{\Omega} V_p(\vec{\omega}) \langle \vec{n}_p, \vec{\omega} \rangle d\vec{\omega} \tag{3.5}$$

describes the amount of arriving light not occluded by surrounding geometry at that point, by the visibility function V_p in all direction of the hemisphere. Naturally, O_p will take a low value inside cavities, as the surface occludes much of the light arriving from the outside. Borland [Bor11] mapped O_p to surface transparency such that cavities become opaque and therefore easily visible. This approach is called *ambient occlusion opacity mapping* (AOOM). A result of this technique is shown in Figure 3.4. Krone et al. [KRS+13] used the AO measure to classify and extract cavity surface patches from triangulated molecular surfaces.

3.3 Solvent Excluded Surface

Connolly [Con83] was the first to present analytical equations to compute the SES, as well as the first algorithm to derive the surface from a dataset of atoms. Geometrically, the surface consists of three basic primitives [Con83; KBE09; KKL+15]:

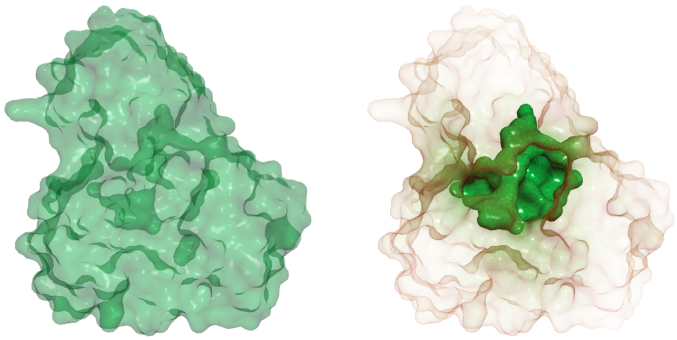

Figure 3.4: Cavity visualizations obtained from ambient occlusion weights. The image was taken from the paper of Borland [Bor11]. Left: Transparency rendering of a molecule. Right: Rendering with AOOM applied. The cavity (green) becomes easily visible.

- *Convex spherical patches*: When the probe is in contact with only one sphere, it traces out the surface of exactly that sphere, leading to a convex spherical patch. The probe has two degrees of freedom in movement.
- *Toroidal patches*: The probe is simultaneously touching two probes, and rolls along both of them. While rolling, the probe's contact position traces a circle on each sphere. The union of all probe position along both spheres forms a torus volume. The toroidal patch is the part of the torus that is located between both contact circles. Here the probe has one degree of freedom.
- *Concave spherical patches* or *spherical triangles*: If the probe is in contact with three sphere at once, it has no degrees of freedom as it is locked in place. The surface is bounded by arcs running between the contact positions.

Additionally, these patches are connected in a certain way: Spherical patches are always connected to toroidal patches, since the probe can transition from rolling with two degrees of freedom to being in contact with another nearby sphere, losing one degree of freedom in return. At the connection of two neighboring toroidal patches, a spherical triangle is located, resulting when the probe is locked in position by three spheres. Therefore, a spherical triangle is always surrounded by three toroidal patches. Figure 3.5 shows an example. Note that in rare cases, the probe can be in contact with more than three spheres at once [Lin10]. These cases can be reduced to multiple spherical triangles with three contacts each. Further, it is possible that an unbroken toroidal patch between σ_i and σ_j exists. This occurs if the probe has no opportunity to come in contact with a third sphere σ_k while rolling along σ_i and σ_j. Another special case are spindle tori [KBE09; Lin10; TA96]. Those occur if the probe intersects the axis of rotation as it rolls around the two connected spheres. In this case the major radius of the traced out torus volume is smaller than the minor radius. Any connected spherical triangles will be *singular*, as they intersect each other.

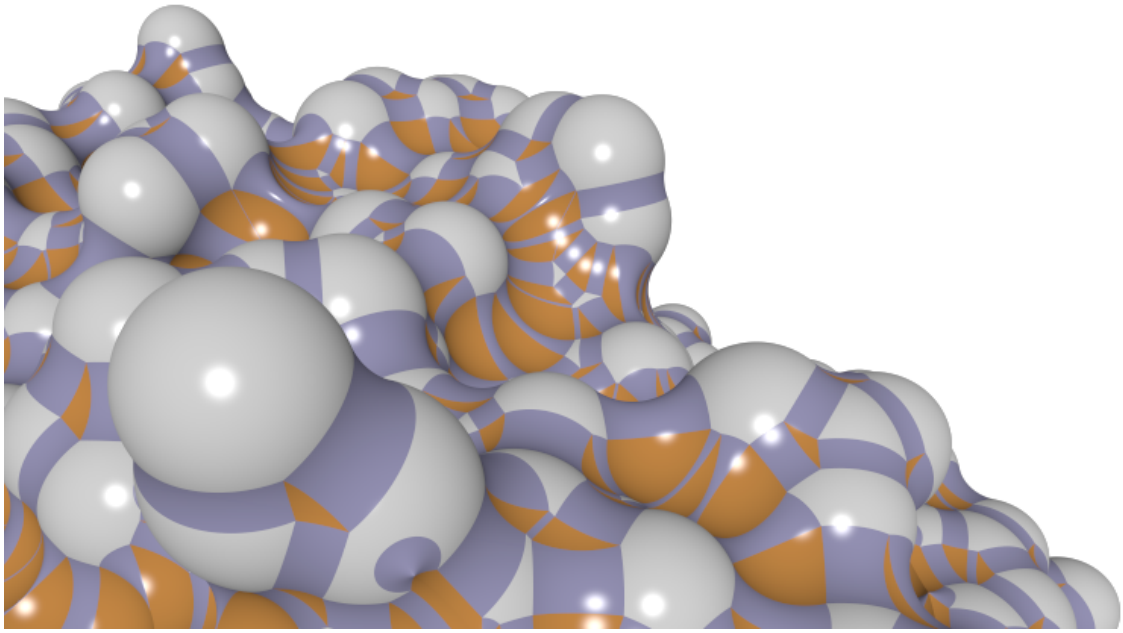

Figure 3.5: Close up view of the SES primitives. Convex spherical patches are white, toroidal patches are purple, while spherical triangles are orange.

Two specific algorithms which compute the surface analytically [SOS96; TA96], as well as a number of discrete approaches were used in recent applications to obtain the SES [CCW06; HKG+17; KBE09; KGE11; LBPH10; Yu09]. These will be discussed in the following.

3.3.1 Reduced Surface Algorithm

The core aspect of the algorithm consists of computation of the *reduced surface* (RS) [SOS96] of the molecule, which can be though of as a graph structure. Afterwards, the SES is derived directly from the RS. Sphere centers serve as graph vertices, called *RS-vertices*. A connection between two vertices is called a *RS-edge*. The circular connection of three edges forms an *RS-face*. Geometrically, RS-edges between σ_i and σ_j indicate that the probe is able to roll while being in contact with σ_i and σ_j, forming a toroidal patch. Therefore, an RS-edge can only exist if σ_i and σ_j are neighbors. The reduced surface is constructed of *RS-faces*. An RS-face indicates that the probe is in contact with three spheres at once, generating a spherical triangle. The existence of an RS-vertex indicates a convex spherical patch. To compute the reduce surface, first an initial RS-Face has to be found [SOS96]. When the molecule is viewed along some axis of the coordinate system, the leftmost surface point of the SES must belong to a spherical patch, and therefore to a valid RS-Vertex. Formally, this can be described as

$$\sigma_{\text{initial}} = \underset{\sigma_i \in S}{\operatorname{argmin}}(p_{i,d} - R_i) \tag{3.6}$$

where $p_{i,d}$ is the d-th component of p_i. After $\sigma_{initial}$ has been found, the unbroken probe path of all neighbors $\sigma_{n_1} \in N(\sigma_{initial})$ is computed, and the sphere whose probe path contains the leftmost point of all probe paths is selected as the second sphere. Now all spheres that are neighbors of $\sigma_{initial}$ and σ_{n_1} are possible RS-faces. Since there are two possible probe position for an RS-face, all those probe position are intersected with S to find the initial RS-face. From this position, as the spherical patch is connected to three toroidal patches, three RS-edges are possible. The probe can roll along each toroidal patch (RS-edge), until the probe comes in contact with a third sphere σ_k. At this point, another RS-face must exist. As it is generally possible that multiple valid third spheres exist that lock the probe in place, the closest one has to be found. This can be achieved by considering the circular path the probe center traces as it rolls to all possible third spheres σ_k, and selecting the one with lowest arc angle, which then forms the next RS-face. This procedure is repeated for each RS-edge that is encountered, until there are no more unprocessed possible RS-edges left. Note that it is possible that the SES consists of multiple unconnected entities, which occur if the probe is unable to transition to at least one other atom sphere without completely losing contact with the all the sphere in S. In this case, the algorithm has to be repeated for all such entities to obtain all *RS-components* [SOS96]. Additionally, the case of an uninterrupted toroidal patch will not generate any RS-face, but a *free RS-edge*. *Free RS-vertices* are analogously generated for any sphere σ_i with $N(\sigma_i) = \emptyset$.

3.3.2 Contour Buildup Algorithm

Some of the notation in the following is borrowed from Lindow [Lin10]. The contour buildup algorithm as introduced by Totrov and Abagyan [TA96] is a per sphere approach to compute the paths that the rolling probe can move along the surface with at most one degree of freedom. Each arc of this representation, called *contour*, represents the path the probe's center point takes when it moves along two spheres σ_i and σ_j of the input dataset. Whenever the probe encounters a third sphere σ_k while rolling along two other spheres σ_i and σ_j, three paths will intersect at this point, as the probe can continue to roll along the path produced by σ_i and σ_k as well as σ_i and σ_j.

By considering each contour of all spheres, the SES is obtained. Figure 3.6 and Figure 3.7 shows examples of the contour and generated surface.

Overview

One important aspect of the algorithm is that during all steps not the radius of σ_i is considered, but its extended radius R_i' which is computed from the probe radius as

$$R_i' = R_i + R_{\text{probe}}, \tag{3.7}$$

which corresponds to the SAS. Consider two spheres σ_i and σ_j where σ_j is contained in the neighborhood set $N(\sigma_i)$. Both extended spheres can be intersected and will produce an intersection circle c_j. Since the outwards exposed surface of the SAS describes all possible center positions of the probe, the intersection circle describes exactly the path the probe

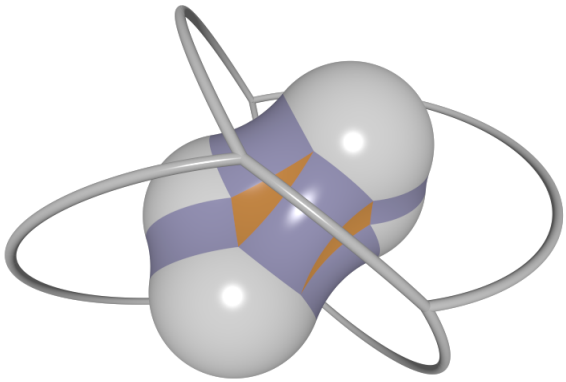

Figure 3.6: Depicted is the contour and resulting surface for a small set of spheres. Convex spherical patches are white, spherical triangles are orange and toroidal patches are purple. Each arc describes the probe's path while being in contact with two spheres. The spherical triangles result from the points where three arcs meet.

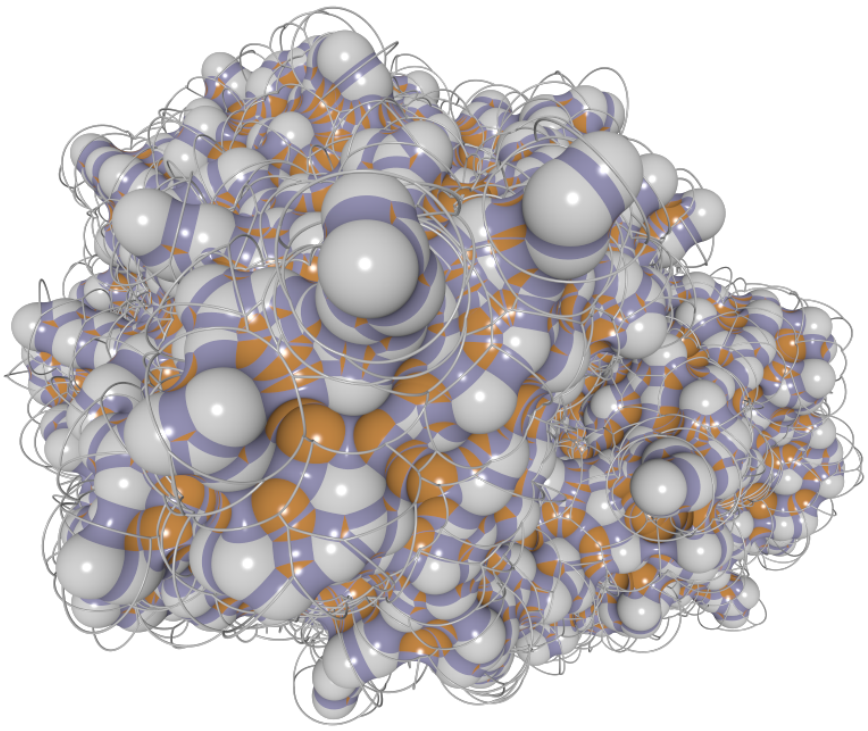

Figure 3.7: Depicted is the contour of the molecule *1rwe* with probe radius 1.4 Å. Convex spherical patches are white, spherical triangles are orange and toroidal patches are purple.

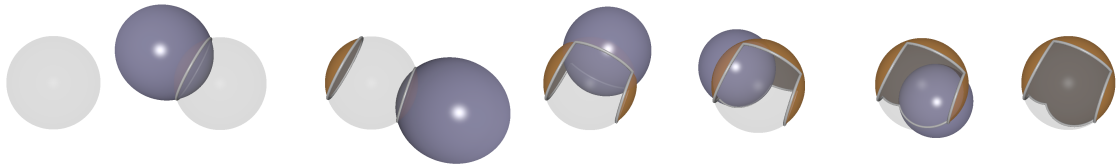

Figure 3.8: From left to right: Neighboring spheres (purple) build the contour of the center sphere (white). On this sphere, all area marked in orange is covered by the neighboring spheres.

center will take when rolling while being in contact with both spheres. Now consider a third sphere σ_k that is also contained in $N(\sigma_i)$, and will produce the intersection circle c_k similarly as before. If c_j and c_k do not intersect, then the probe cannot switch from rolling between σ_i and σ_j to rolling between σ_i and σ_k. However if they do intersect, then there are up to two possible positions where the probe can switch from being in contact with σ_j to σ_k, and vice versa. Intersecting a circle with another circle splits each one into two arcs. One of these arcs of each circle is always discarded, as it lies within the extended sphere that belongs to the intersecting circle. Repeatedly, the remaining neighboring spheres σ_l are intersected to obtain their intersection circles c_l. All current arcs can then be checked for modification (shortening or splitting), which might break the contour open. In such a case, the contour has to be repaired by considering where on c_l old arcs end and begin, and generating new arcs to close any holes. Note that the neighborhood set $N(\sigma_i)$ still concerns the unextended radii and is therefore unchanged. If $\sigma_j \in N(\sigma_i)$ holds the extended spheres intersect.

Computation of the contour with regard to the sphere σ_i involves two phases [KGE11; Lin10]:

- Computation of all relevant intersection circles C of neighboring spheres $\sigma_j \in N(\sigma_i)$ with extended radii R_i' and R_j'.
- Computation of the arcs of the contour by considering each circle from C and determining how it may change the current contour.

Note that it is not required to first compute all circles and then the arcs afterwards, each circle can be processed and compared to the current arcs directly, as described by Totrov and Abagyan [TA96]. However, certain circles (or even the entire sphere) are eliminated after the first step, which is simplifying arc management in the second step. Figure 3.8 shows the stepwise buildup of the contour for a single sphere.

Circle Computation

The first phase of the algorithm computes all relevant intersection circles between σ_i and all $\sigma_j \in N(\sigma_i)$. Let C_n contain all current relevant circles of σ_i at the time of processing c_j, where already n circles have been processed, and $n \in \{0, ..., |N(\sigma_i)| - 1\}$. Therefore $C_0 = \emptyset$ denotes the first set of circles. Additionally, let $\hat{C}_n$ contain all those circles found to be

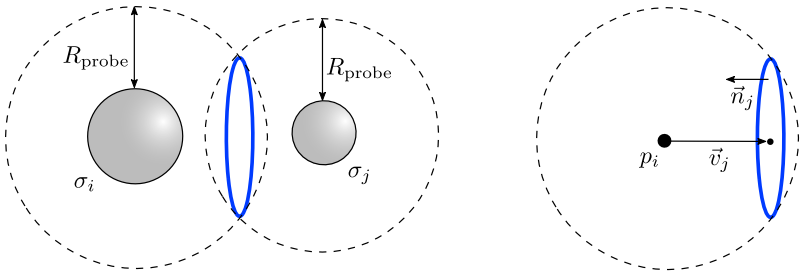

Figure 3.9: Example of two spheres σ_i and σ_j with extended radius (dotted spheres) being intersected, which yields the blue circle defined by the circle center $\vec{v}_j$ relative to p_i.

removed while processing c_j in step n. Again consider two spheres σ_i and its neighbor σ_j, and their intersection circle c_j. The circle's center position vector is computed as [TA96]

$$\vec{v_j} = \vec{v}_{i,j} \frac{R_i^2 + \langle \vec{v}_{i,j}, \vec{v}_{i,j} \rangle - R_j^2}{2\langle \vec{v}_{i,j}, \vec{v}_{i,j} \rangle}. \tag{3.8}$$

with $\vec{v}_{i,j} = p_j - p_i$, local to σ_i's position p_i. Additionally, the circle's normal is defined as $\vec{n}_j = -\vec{v}_{i,j}$ [Lin10]. See Figure 3.9 for a depiction. The circles radius [Lin10] is then

$$r(c_j) = \sqrt{R_i'^2 - \langle \vec{v}_j, \vec{v}_j \rangle}. \tag{3.9}$$

Now c_j is checked against all $c_k \in C_n$ for intersection. Let $\vec{v}_k$ be the position vector of the center of c_k. If the circles c_j and c_k intersect, the planes containing them must also intersect, and the corresponding intersection line must pass though both circles. This is only the case if the line enters the extended sphere of σ_i at first circle intersection and leaves at the second one. A third plane positioned at the origin, containing both $\vec{v}_j$ and $\vec{v}_k$ can be constructed and intersected with the line, yielding the auxiliary position vector [TA96]

$$\vec{h} = \frac{\vec{v}_j(\langle \vec{v}_j, \vec{v}_j - \vec{v}_k \rangle \cdot \langle \vec{v}_k, \vec{v}_k \rangle) + \vec{v}_k(\langle \vec{v}_k, \vec{v}_k - \vec{v}_j \rangle \cdot \langle \vec{v}_j, \vec{v}_j \rangle)}{\langle \vec{v}_j, \vec{v}_j \rangle \cdot \langle \vec{v}_k, \vec{v}_k \rangle - (\langle \vec{v}_j, \vec{v}_k \rangle)^2}. \tag{3.10}$$

This position always lies at equal distance to both intersection points if they exist, the corresponding position vectors is computed as

$$\vec{x}_{1,2} = \vec{h} \pm \vec{a} \cdot \frac{R_i'^2 - \langle \vec{h}, \vec{h} \rangle}{\langle \vec{a}, \vec{a} \rangle}, \tag{3.11}$$

where $\vec{a} = \vec{v}_j \times \vec{v}_k$ [TA96]. Figure 3.10 shows an example. The intersections are not required here, but they become relevant in the second phase of the algorithm. The vector $\vec{h}$ is used to determine if c_j and c_k are intersecting which is the case if $\vec{h}$ lies inside of the extended sphere of σ_i, i.e. if $\langle \vec{h}, \vec{h} \rangle \leq R_i'^2$ holds. If there is an intersection, c_k and c_j are positioned such that they may contribute with arcs to the contour.

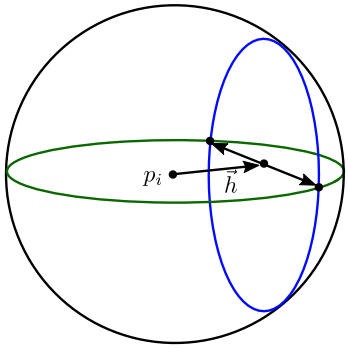

Figure 3.10: Two circles that intersect produce the halfway vector $\vec{h}$, which is located between the intersections.

Otherwise, the four cases depicted in Figure 3.11 may occur:

1. Sphere σ_k and sphere σ_j do not cover c_j and c_k respectively: No interaction.
2. Sphere σ_k covers circle c_j completely: c_j is added to $\hat{C}_{n+1}$. Processing for c_j may stop.
3. Sphere σ_j covers circle c_k completely: c_k will not contribute to the contour and can be removed entirely, i.e. c_k is added to $\hat{C}_n$.
4. Sphere σ_j covers circle c_k *and* sphere σ_k covers circle c_j: Sphere σ_i is completely covered by σ_j and σ_k, there is no contour for σ_i. Processing may be stopped for σ_i.

In this context, sphere σ_j *covers* circle c_k if all points on the circle $u \in c_k$ are inside the extended radius of σ_j. Lindow [Lin10] proposed to determine the case from three quantities

$$\begin{aligned} g_1 &= \langle \vec{n}_j, \vec{n}_k \rangle, \\ g_2 &= \langle \vec{m}_j, \vec{m}_k \rangle, \\ g_3 &= \langle \vec{n}_j, \vec{q} \rangle, \end{aligned}$$

where $\vec{m}_i = \vec{v}_i - \vec{h}$ and $\vec{q} = \vec{v}_k - \vec{v}_j$.

If c_j and all c_k are either intersecting or not completely cut away, they appear in C_{n+1}, i.e.

$$C_{n+1} = (C_n \cup \{c_j\}) \setminus \hat{C}_n. \tag{3.12}$$

Finally after all c_j have been processed, $C_{|N(\sigma_i)|}$ contains all relevant circles that may contribute with arcs in the contour. Algorithm 3.1 gives an algorithmic description of the circle computation.

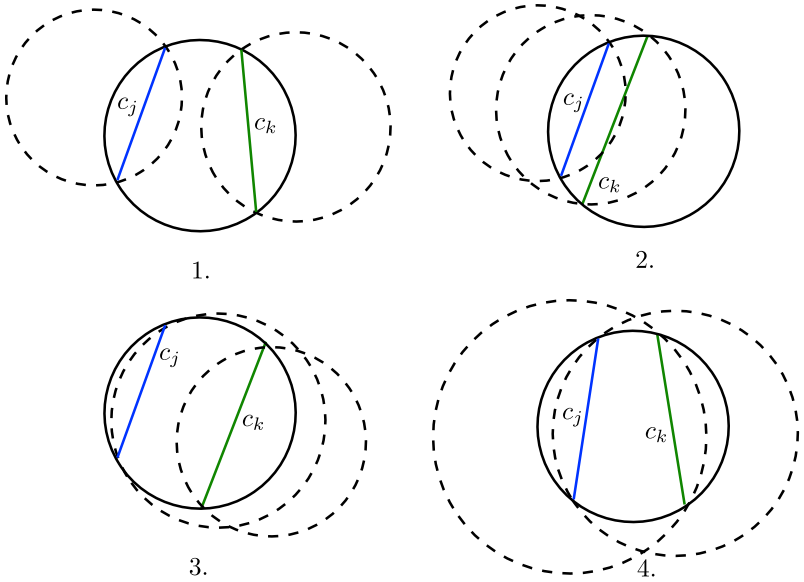

Figure 3.11: Depiction of the four cases the circles c_j and c_k may be in if there is no intersection between them. In the first case, the circles are not covered. The second case occurs if c_j is covered by the sphere of c_k. Analogously, the third case happens if c_k is covered by c_j. If both circles are simultaneously covered, the entire sphere is covered.

Table 3.1: The quantities that describe the mutual configuration of two circles. T stands for true, F for false.

$g_1 > 0$	$g_2 > 0$	$g_3 > 0$	Case
T	T	T	2
T	T	F	3
T	F	T	1
T	F	F	4
F	T	T	1
F	T	F	4
F	F	T	2
F	F	F	3

Contour Computation

Let $C = C_{|N(\sigma_i)|}$ for readability. Similarly to the previous section, A_n is the arc set of step n of the algorithm for sphere σ_i, and $n \in \{0, ..., |C| - 1\}$. Further, a_k denotes some arc constructed from circle c_k. Note that there may be more than one a_k for a circle c_k. Again, $A_0 = \emptyset$ is the initial set of arcs. Additionally, $\vec{s}(a_k)$ describes the position vector pointing towards the start point of arc a_k and $\vec{e}(a_k)$ to its end point, relative to p_i. In the

Algorithm 3.1 Circle computation for sphere σ_i

```
procedure ComputeCircles(σ_i, N(σ_i), R_probe)
    C_0 ←∅
    for n = 0, ..., |N(σ_i)| − 1 do
        c_j ←ComputeCircle(σ_i, σ_j, R_probe)
        for all c_k ∈ C_n do
            if c_j and c_k do not intersect then
                Circlecase ←ComputeCircleCase(c_j, c_k)
                if Circlecase = 1 then
                    Do nothing
                else if Circlecase = 2 then
                    Ĉ_n ←Ĉ_n ∪ {c_j}                    // c_j is covered
                    Break
                else if Circlecase = 3 then
                    Ĉ_n ←Ĉ_n ∪ {c_k}                    // c_k is covered
                else                                    // Circlecase = 4
                    return σ_i entirely covered
                end if
            end if
        end for
        C_{n+1} ←(C_n ∪ {c_j}) \ Ĉ_n
    end for
    return C_{|N(σ_i)|}
end procedure
```

following, arcs are defined to run from $\vec{s}(a_k)$ to $\vec{e}(a_k)$ in clockwise direction observed from σ_k. From C all arcs are computed by considering every circle $c_j \in C$ and intersecting it against the current contour. In step n, the current circle c_j is processed. Every currently present arc $a_k \in A_n$ has to be compared with c_j. The arc a_k is part of the circle c_k, which was produced by intersection with sphere σ_k. First circle c_j is intersected with circle c_k to find possible intersections by evaluating Equation 3.11 to obtain $\vec{x}_{1,2}$. Note that x_1 and x_2 have to be exchanged for each other, if $\langle p_j - p_i, \vec{v}_j \rangle < 0$ [Lin10]. Depending on the positions of $\vec{s}(a_k)$, $\vec{e}(a_k)$ and $\vec{x}_{1,2}$, the arc is

- Not modified, which happens when a_k is outside of σ_j.
- Shortened, either at $\vec{s}(a_k)$ or $\vec{e}(a_k)$, if a_k is partially engulfed by σ_j.
- Split, if $\vec{s}(a_k)$ or $\vec{e}(a_k)$ are outside of σ_j, but the arc's bend is covered by σ_j.
- Eliminated, if all points of a_k are covered by σ_j.

Let $\hat{A}_n$ contain all modified arcs of step n, and E_n the corresponding arc endpoints that were changed. Additionally $\bar{A}_n$ denotes all unmodified arcs of step n, and $\tilde{A}_n$ all arcs that are newly created. Two different types of intersection of σ_i with the sphere σ_j can occur:

- The *touching* case: σ_i and σ_j intersect in a small region, i.e. $\langle \vec{v}_j, \vec{n}_j \rangle < 0$
- The *engulfing* case: σ_j almost completely covers σ_i, i.e. $\langle \vec{v}_j, \vec{n}_j \rangle > 0$

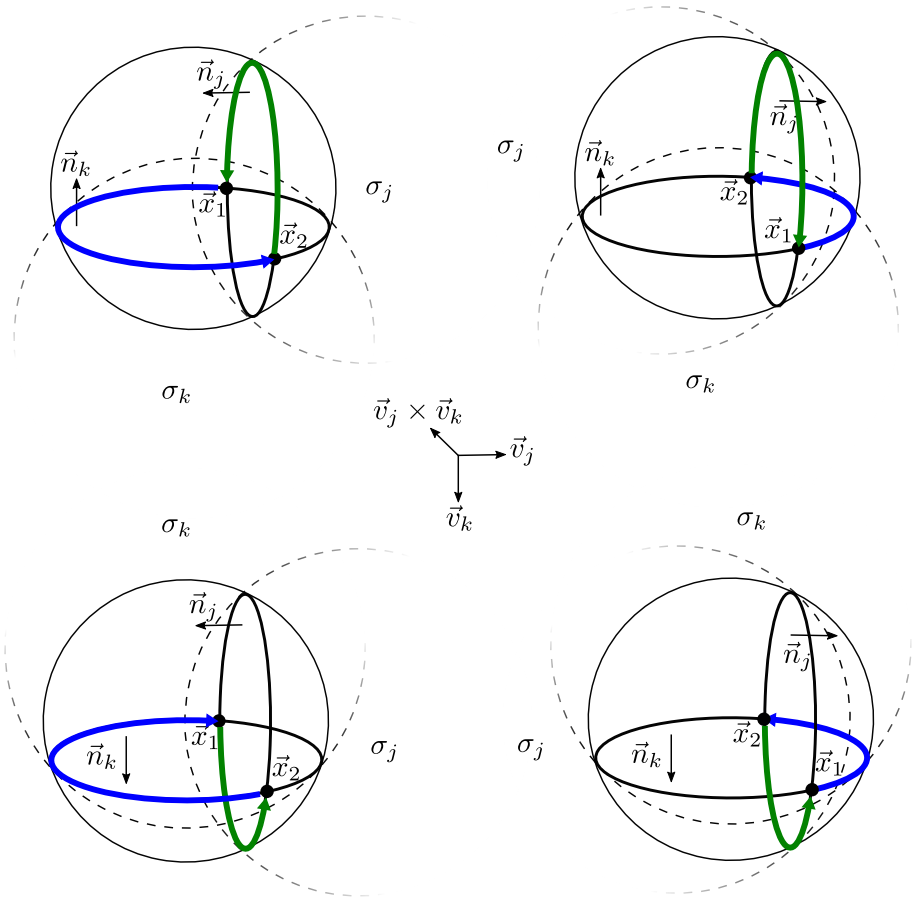

Figure 3.12: The four possible situations that may occur around the sphere σ_i. Top-left: Both spheres are touching σ_i. Top-right: σ_j is engulfing while σ_k is touching σ_i. Lower-left: σ_j is touching while σ_k is engulfing σ_i. Lower-right: Both spheres are engulfing σ_i. The blue and green arcs are the leftover parts of the circles c_k and c_j. Note that the vector $\vec{v}_j \times \vec{v}_k$ points from $\vec{h}$ towards $x_{1,2}$.

Three geometric quantities are required to determine how the arc a_k is to be processed [Lin10; TA96]:

$$d_1 = \langle p_j - p_i, \vec{s}(a_k) - \vec{v}_j \rangle, \tag{3.13}$$
$$d_2 = \langle p_j - p_i, \vec{e}(a_k) - \vec{v}_j \rangle, \tag{3.14}$$
$$d_3 = \langle \vec{v}_l, p_k - p_i \rangle \cdot \langle \vec{s}(a_k) \times \vec{x}_1, \vec{e}(a_k) \rangle, \tag{3.15}$$

These quantities can be used to identify if and how a_k is to be modified, depending on the mutual configuration of involved spheres as seen in Figure 3.12.

The quantity d_1 describes the position of $\vec{s}(a_k)$ relative to the intersection circle c_j. If $d_1 < 0$, then $\vec{s}(a_k)$ is not cut away by σ_j, otherwise, this start point is engulfed. Similarly d_2 describes $\vec{e}(a_k)$ situation. The first dot product of d_3 produces a negative sign in the engulfing case and a positive sign in the touching case. Figure 3.13 shows and example were all used vectors are depicted. Further, the second dot product constructs a plane

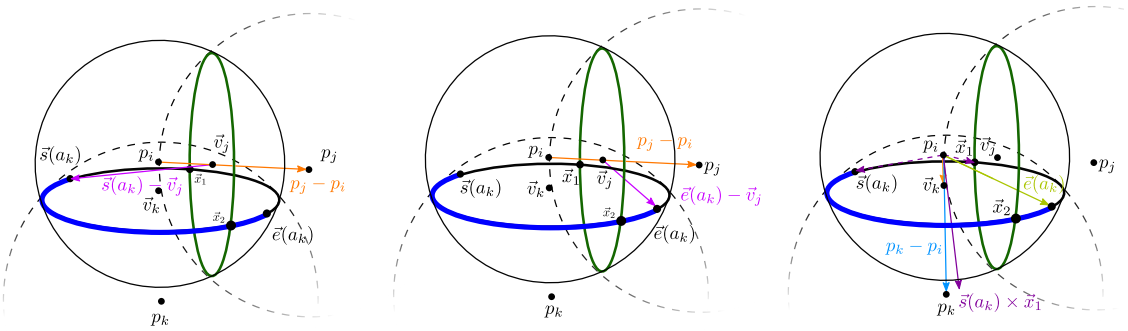

Figure 3.13: Example depiction of arc modification of the arc a_k (blue), corresponding to its circle c_k, which is intersected by circle c_j (green). From left to right: Vectors used in computation of d_1, d_2 and d_3, respectively. Here $d_1 < 0$, $d_2 > 0$ and $d_3 > 0$. Therefore, the arc start remains the same, and the arc end $\vec{e}(a_k)$ is shortened to $\vec{x}_2$, as $\vec{e}(a_k)$ is inside the extended sphere of σ_j.

Table 3.2: Arc modification based on the three quantities. T stands for true, F for false.

$d_1 > 0$	$d_2 > 0$	$d_3 > 0$	Operation
T	T	T	Arc removed
T	T	F	Arc is shortened
T	F	T	$\vec{s}(\hat{a}_k) = \vec{x}_2$ and $\vec{e}(\hat{a}_k) = \vec{e}(a_k)$
T	F	F	$\vec{s}(\hat{a}_k) = \vec{x}_1$ and $\vec{e}(\hat{a}_k) = \vec{e}(a_k)$
F	T	T	$\vec{e}(\hat{a}_k) = \vec{x}_2$ and $\vec{s}(\hat{a}_k) = \vec{s}(a_k)$
F	T	F	$\vec{e}(\hat{a}_k) = \vec{x}_1$ and $\vec{s}(\hat{a}_k) = \vec{s}(a_k)$
F	F	T	Arc unchanged
F	F	F	Arc split in two.

containing $\vec{s}(a_k)$, $\vec{x}_1$ and the origin, and produces a sign depending on the side $\vec{e}(a_k)$ is on. In some cases depending on the sign of $d_4 = \langle \vec{n}_k < \vec{v}_k \rangle$ arcs are modified differently. Let $\hat{a}_k$ be the modified arc. Arc shortening happens in the following way: If $d_4 < 0$ then $\vec{s}(\hat{a}_k) = x_1$, $\vec{e}(\hat{a}_k) = x_2$. Otherwise $\vec{s}(\hat{a}_k) = x_2$, $\vec{e}(\hat{a}_k) = x_1$. If the arc is split in two, another arc a'_k is created. It is always the case that

$$\vec{e}(a'_k) = \vec{e}(a_k),$$
$$\vec{s}(\hat{a}'_k) = \vec{s}(a_k)$$

since both start and end point of the arc are preserved. If $d_4 < 0$ then

$$\vec{e}(\hat{a}_k) = \vec{x}_2,$$
$$\vec{s}(a'_k) = \vec{x}_1.$$

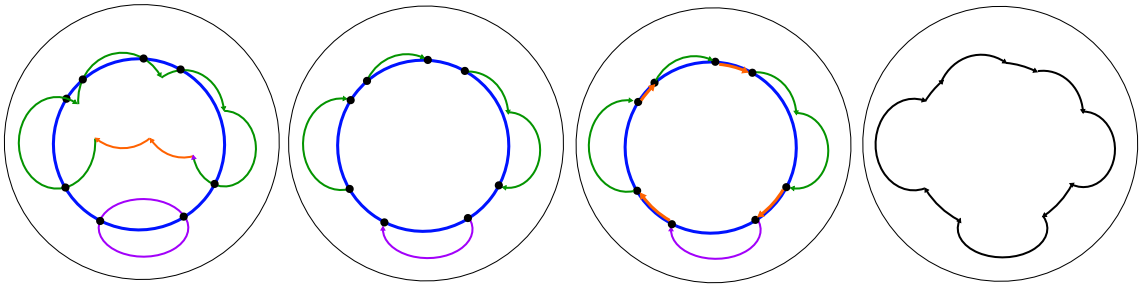

Figure 3.14: Example of the contour repair procedure. First, the blue circle c_j is intersected with all arc circles (black dots) and classified: Green arcs are modified and orange arcs are removed. The purple arc consists of a whole circle. Next, the arcs are modified and the endpoints on the blue circle are saved. After the points are sorted, one moves clockwise around the circle and connect end to start points to obtain new arcs. Finally, the contour is repaired and the next circle can be processed.

Otherwise the modification is done as

$$\vec{e}(\hat{a}_k) = \vec{x}_1,$$
$$\vec{s}(a'_k) = \vec{x}_2.$$

If $\hat{A}_n \neq \emptyset$, the contour has to be repaired. Modified arc endpoints E_n have to be connected again, if σ_j broke the contour by consuming arcs partially (shortening, splitting) or completely (removal). Since all modified endpoints in E are located on c_j, corresponding new arcs must be constructed from c_j as well. Let there be M modified arc points $O = \{\alpha_0, ..., \alpha_m, ..., \alpha_{M-1}\}$, ordered clockwise (seen from σ_j) according to their position on c_j, i.e. α_m is ordered before α_{m+1}, as α_m is located in clockwise order before α_{m+1} on c_j. Note the cyclic nature, i.e. α_{M-1} is ordered before α_0. Then a new arc from α_m to α_{m+1} (in clockwise direction) must be created if α_m is an endpoint and α_{m+1} is a start point, to fill this broken part of the contour. Figure 3.14 depicts the repairing of the contour. If no arcs were modified by c_j, then c_j can be added as a whole to the contour, but only if there is not intersection of c_j with any of the circles that were processed before c_j. This means the next arc set A_{n+1} is computed as

$$A_{n+1} = \hat{A}_n \cup \bar{A}_n \cup \tilde{A}_n. \tag{3.16}$$

Completing this procedure for all circles in C will result in the final contour of σ_i being contained in $A_{|C|}$. An algorithmic description can be found in Algorithm 3.2.

Remarks

For the implementation of the algorithm, it is important to flag spheres according to their state, i.e. if they were completely covered in the first phase (also referred to as *buried*). An empty circle set C does not indicate this as it is possible that a sphere has no neighbors. Additionally, the arc set $A_{|C|}$ might be empty, which also indicates that σ_i is covered. If

Algorithm 3.2 Contour computation for sphere σ_i from circles C

```
procedure ComputeContour(σ_i, C, R_probe)
    A_0 ←∅
    for n = 0, ..., |C| − 1 do
        Â_n ←∅                                              // Modified arcs
        E_n ←∅                                              // Modified arcs endpoints
        Ā_n ←∅                                              // Unmodified arcs
        Ã_n ←∅                                              // New arcs
        c_j ←n-th circle of C
        for all a_k ∈ A_n do
            c_k ←Circle of arc a_k
            if c_j and c_k do not intersect then
                Ā_n ←Ā_n ∪ a_k
                Continue with next arc a_k
            end if
            x⃗_{1,2} ←ComputeIntersection(c_j, c_k)
            Update Â_n, E_n and Ã_n as arcs are modified or created
            O ←SortArcEndsClockwise(c_j, E_n)
            for all α_m ∈ O do                              // Iterated in clockwise order
                if α_m is an arc end then
                    Create arc that connects α_m with α_{m+1} and add to Ã_n
                end if
            end for
        end for
        if There was no intersection of c_j with any a_k ∈ A_n then
            if No previously processed c'_j ∈ C intersects c_j then
                Ã_n ←Ã_n∪ ToArc(c_j)                        // Add whole circle as arc
            end if
        end if
        A_{n+1} ←Â_n ∪ Ā_n ∪ Ã_n
    end for
    return A_{|C|}
end procedure
```

the contours of all N spheres are computed, each arc will appear twice, as it is computed for each sphere separately. This has to be considered when the SES is to be generated from the contour. The neighbor set $N(\sigma_i)$ can be efficiently computed by using a spatial data structure, most commonly a grid [KBE09; LBPH10; Lin10] since it is fast to construct and offers constant lookup time.

Numerical Issues

The computation of the measures d_1, d_2 and d_3 involve dot products which can produce values arbitrarily close to zero. In the face of numerical inaccuracy, if one of these measures is very close to zero, no proper decision can be made how to modify an arc, which is called

a *singularity* [Lin10; TA96] (not to be confused with the singularities occurring between spherical triangles). This occurs when more than three spheres are in contact with the probe simultaneously. Totrov and Abagyan [TA96] suggest two possible approaches to deal with such a case:

1. The complete contour is omitted for all spheres that are involved in the singularity.
2. Spheres involved in a singularity are moved slightly in random directions and afterwards the contour is recomputed for all spheres that are either singular or neighbors of a singular sphere (or now became a neighbor of a moved sphere).

The first approach will lead to holes in the contour, however it is computationally trivial to achieve. If a closed surface is absolutely required, the second approach may has to be repeated arbitrarily often, as new singularities can occur by moving the current singular spheres, leading to high computational effort [Lin10].

Generating the Surface

Convex sphere primitives are generated if the corresponding sphere σ_i was found to not being covered in the previous steps. Toroidal patches are generated from the arcs of the contour. Recall that the computed arcs represent the paths the probe can take while rolling with less than two degrees of freedom along the spheres. Let A be the set of arcs of σ_i's contour. Further, $a_j \in A$ denotes an arc of the contour, describing the probe's path while being in contact with σ_i and σ_j. An arc of the contour generates a toroidal patch with major radius equal to the corresponding circle's radius, minor radius R_{probe} [Lin10; TA96] and center point $p_i + \vec{v}_j$. As mentioned before, if two spheres are neighbors, both their arc sets will contain the same arc. Therefore a_j should only produce a toroidal patch if $i < j$. Spherical triangles are generated from the arc end points where at least three arcs meet, with sphere radius R_{probe}. Again let $a_j \in A$ be an arc of sphere σ_i and also let a_k be the successor arc of a_j, i.e.

$$\vec{e}(a_j) = \vec{s}(a_k), \tag{3.17}$$

both corresponding to neighboring spheres σ_j and σ_k respectively. Then the spherical triangle at $p_i + \vec{e}(a_j)$ should only be created if $i < j$ and $i < k$ (similar to Lindow's approach [Lin10]). If a spherical triangle is connected to a toroidal patch whose minor radius is greater than the major radius, then it will protrude into the other connected spherical triangle, producing a singularity (different to the contour singularities). It is even possible that spherical triangles just being in the vicinity are close enough together to produce such a singularity. Singularities can be cut away by intersecting the spherical triangle with all probe spheres positioned around it that intersect the spherical triangles probe sphere.

3.3.3 Discrete Algorithms

The solvent excluded surface can be computed not only analytically but also in discretized space. Can et al. [CCW06] presented a two-phase algorithm to compute a discretized representation of the SES. In the first phase, the SAS is computed by propagating a

outward front from every atom until its extended volume is covered spatially for each atom. Then in a second step from the SAS an *inward front* covering the probe radius is propagated. Wherever the second propagation front stops lies the SES. The authors carefully ensured that grid cells are visited at most once per phase. A similar, but list based method was proposed by Yu [Yu09]. Rendering of these representation is done with volume rendering approaches. The approach by Hermosilla et al. [HKG+17] computes a grid based distance field that contains approximate, local positive scalar values outside the surface, and negative values inside the surface, allowing rendering similar to sphere tracing [Har96]. Distances are only considered in a small area around the grid points. First grid points are classified according to their position, i.e. completely outside of the SES, inside the SES (grid point is inside of an atom) or on the boundary of the SES (grid point is located between the SAS and van der Waals surface). In a second phase, all boundary grid points are assigned accurate distance by inspection of its neighboring grid points for inside grid points. An advantage of this method is the support for progressive refinement and suitability for GPU implementation.

Such algorithms might be able to compute the SES faster than the classical analytical approaches in certain situations, but suffer from high memory usage depending on the grid size. The computed solutions will also never be as accurate as the ones obtained from analytical methods. This becomes apparent since for example small cavities can be missed entirely.

3.3.4 Rendering Transparency

Most current approaches [JPSK16; KBE09; KFR+11; KKP+13; LBPH10] to render the SES use ray casting on GPUs to render the primitives. Since this is a object-order approach, objects must be rendered in a sorted order, from front to back or back to front, to achieve correct blending. Back to front compositing works by repeatedly blending fragments starting at the most distant fragment with the blending equation [BM08]

$$C_{\text{dst}} \leftarrow A_{\text{src}} C_{\text{src}} + (1 - A_{\text{src}}) C_{\text{dst}}, \tag{3.18}$$

where C_{src} and A_{src} refer to the current fragments color and opacity (alpha) values, and C_{dst} is the current computed color. Similarly the fragments can be blended in front to back order by repeatedly applying the equations [BM08]

$$C_{\text{dst}} \leftarrow A_{\text{dst}} (A_{\text{src}} C_{\text{src}}) + C_{\text{dst}}, \tag{3.19}$$

$$A_{\text{dst}} \leftarrow (1 - A_{\text{src}}) A_{\text{dst}}. \tag{3.20}$$

Note that here A_{dst} describes transparency, not opacity, and is initialized to $A_{\text{dst}} = 1$. Back to front compositing is sometimes referred to as *OVER* blending, while the front to back approach is called *UNDER* blending [Dun14]. If ordering is to be avoided, approximate schemes based on weighted sums as the one proposed by McGuire and Bavoil [MB13] can be used. Such methods are less relevant for this work, as ray tracing implicitly gives the correct order of intersections. This can be achieved by presorting all primitives and sending them in this order to the graphics hardware. However, since this operation can be computationally costly, another popular approach is *depth peeling* [BM08; EW01]. This algorithm renders

the scene layer by layer. During each step, the previously rendered surface is rejected based on its depth values, such that the next closest surface can be rendered. The layers are then blended together. The algorithms main disadvantage is the requirement for N rendering passes, if the surface is N layers deep. However, rendering can be stopped after a fixed number of steps since the more a surface is occluded by other surfaces, the less its impact will be in the final image [EW01]. Ray tracing on the other hand simplifies ordering as ray traversal implicitly gives the order by the intersections that are encountered along the way. However, traversal of tree based spatial data structure like a BVHs does not always return intersection in the correct order if tree nodes overlap. Amstutz et al. [AGGW15] showed that it is possible to obtain the correct order of intersections by exploiting the fact that intersections belonging in order are encountered closely together and require few operations to obtain correctly sorted intersections. The authors proposed using insertion sort whenever a new intersection is encountered, or selection sort as a post process on all encountered intersections, and tested different data layouts for employed data structures (AoS versus SoA, see Section 2.4.1). Their tests revealed that CPU ray traversal benefits from using the AoS layout and post traversal selection sort, while the GPU implementation achieved best performance using SoA layouts and the insertion sort technique.

Solvent excluded surfaces have been rendered before with transparent primitives [JPSK16; KKP+13]. The main difficulty is culling of geometry that lies inside the surface, while the exposed parts are left intact. While spherical triangles do not leave parts behind, toroidal patches and convex spherical patches do. Inner toroidal patch can be cut by four cutting planes, or by the visibility sphere and two planes. Details are given in Section 4.4.2. One GPU-based approach was introduced by Kauker et al.[KKP+13], which uses per-pixel arrays or per-pixel linked lists to render transparency, called the *puxels* algorithm. Per-pixel arrays require two render passes to determine the number of items per pixel, and afterwards the items themselves. First all fragments are collected in unordered per-pixel arrays by rendering all primitives and saving for each fragment at least its depth value and color. Now these lists are sorted according to their depth values to obtain ordered fragments per pixel. Afterwards, a *constructed solid geometry* (CSG) [SM09] approach is used to cut away the inner parts of the computed primitives. This is achieved by placing the RS as additional geometry inside the SES, and rendering both objects with the CSG *union* operation. Toroidal patches are rendered as closed objects. The union of all objects completely covers the interior space of the SES. Further, the union operation is implemented by using a counter that is incremented when front facing geometry is hit, and decremented for back facing geometry. Only such surfaces are rendered that cause the counter to increase to one while front facing geometry is encountered, or decrease the counter to zero while back facing geometry is encountered. At last, the surviving fragments are blended in correct order to obtain the final image. Jurcik et al. [JPSK16] proposed another GPU-based approach that implements transparent SES. Similar to the approach by Kauker et al. [KKP+13], the authors used pixel linked lists to render fragments which are subsequently sorted and blended. However, they did not use the RS surface but used bounding geometry derived from the connected toroidal patches, though they do not give further details.

4 Implementation

This chapter describes the contour buildup implementation and used optimizations. Additionally, a possible new approach for the computation of the SES is outlined. Further, the implementation of the different surface primitives is described in detail. In the last section, the computed surface is used to implement a real time variant of the AOOM method (see Section 3.2). The contour buildup algorithm was chosen over the other methods as it is easily parallelizable over the input spheres, as there are no dependencies between spheres when the contour is computed [KGE11; LBPH10]. In contrast, the RS-algorithm is sequential in nature. Principally it would be possible to start the algorithm at multiple initial spheres, however stopping the algorithm such that the entire reduced surface is computed without leaving holes *and* ensuring that no or only a low number of surface elements are computed multiple times would imply severe synchronization overhead. One possible drawback of the contour buildup algorithm is the fact that every contour element is computed twice. Despite this, Lindow et al. [LBPH10] showed that the Contour Buildup is comparable in speed to the RS-algorithm, even in single core execution. While the OSPRay Framework aims for classical ray tracing, a discretized approach like the one presented by Hermosilla et al. could be implemented as well. However as mentioned before, such approaches suffer from inaccuracies depending on the discretization level, while the analytical approaches will inherently be correct in the sense that no surface features are missed. In addition, the analytical approaches allow the generation of explicit primitives such that Embree's highly optimized bounding volume hierarchy can be used straightforwardly.

4.1 Contour Buildup

The CPU implementation was done according to Section 3.3.2 in C++. A number of specific issues have to be considered. Input of the algorithm is given as the datatype shown in Listing 4.1. The type `vec3r` refers to a vector of three `real` numbers (`double` or `float`).

During computation, it might happen that Equation 3.10 becomes singular if the denominator vanishes. In this case, the computation for the affected sphere is stopped.

Listing 4.1 The sphere struct used in the implementation.

```
1    struct Sphere {
2        vec3r p;
3        real R;
4    };
```

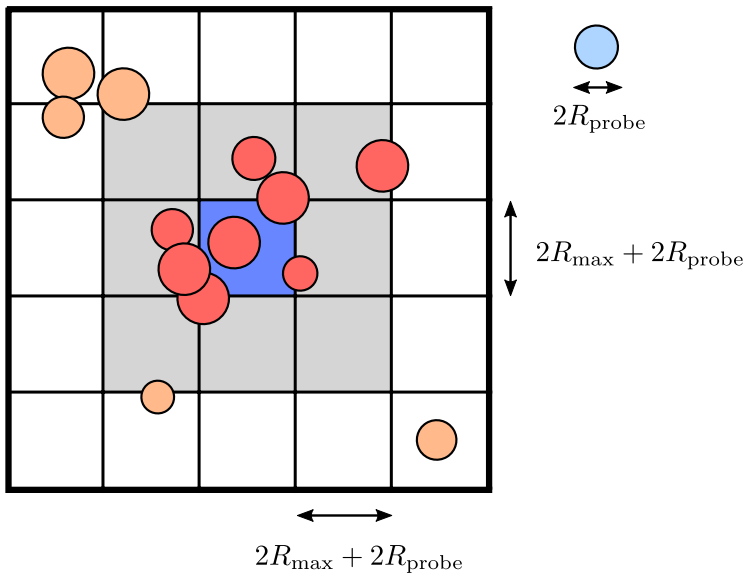

Figure 4.1: Example grid for the probe (blue sphere) with radius R_{probe} for neighborhood computation with the approach described by Krone et al. [KBE09]. For the blue grid cell, the spheres whose center is located in the gray cells (red) have to be distance checked. All other spheres are guaranteed to be out of range.

4.1.1 Neighborhood Computation

In order to do the circle computation described in Section 3.3.2, the neighbor set of the current sphere has to be determined. Grids have been employed for this task previously [KBE09; LBPH10; Lin10]. In general for each sphere σ_i its position p_i locates the sphere's cell. A lookup into a cell returns all spheres whose position is inside the cell. Lindow [Lin10] used a fine grid whose cell size is determined heuristically. This requires iterating over all cells that may contain neighboring spheres, which will resemble a voxelized sphere. The implementation of this work uses the same approach as Krone et al. [KBE09], which consists of choosing the cell length such that all neighbors of a sphere are found by examining the sphere's own cell and all eight neighboring cells. The cell side lengths are uniformly chosen as

$$2R_{\text{max}} + 2R_{\text{probe}}, \tag{4.1}$$

where R_{max} is the maximum sphere radius [KBE09]. This ensures that all possible neighbors are located in a 3×3 sub grid. Figure 4.1 shows a depiction of such a grid. Note that for each of the spheres in the sub grid another range check has to be done to obtain the final neighbor set. Every cell contains the indices of the contained spheres in a `std::vector`. `std::vector` is a C++ standard library container that allows constant access to elements but may reallocate itself if its current memory is too small[1]. Note that with large grids this is not optimal as then many `std::vector` instances are allocated, each with its own management data generating memory overhead. However, this allows grid construction

[1] See the C++ standard, https://isocpp.org/.

Listing 4.2 The circle struct used in the implementation.

```
1   struct Circle {
2       vec3r v;
3       vec3r n;
4       real radius;
5       int sphereIdx;
6       bool buried;
7   };
```

in $O(N)$, where N is the number of spheres. In principle it is possible to allocate a single `std::vector` and write all indices in continuous memory as an indices list. Each cell would need to track how many sphere it contains and manage an index into the indices list. However, in this case the effort would be at least $O(X \times Y \times Z)$, as then the grid cells in each dimension X, Y and Z would have to be iterated. Additionally, it is not trivial to then find all the spheres that are located in a specific grid cell, as this is the intend of the grid in the first place.

4.1.2 Circle Computation

After the neighbors of a sphere σ_i are known, the circle computation can begin (Section 3.3.2). Usually most computation effort is spend in this phase of the algorithm, as shown in Section 5.1. Listing 4.2 shows the employed circle datatype. In the `Circle` struct, `sphereIdx` refers to the sphere that created the circle. The `buried` flag indicates if a circle is covered completely by another circle's extended sphere. The resulting circle list is written to a `std::vector`. Since it is possible that circles become buried and therefore no longer relevant, they can be replaced by new circles. This is advantageous as this means that the final circle list is shorter and also less prone to expensive reallocation if the `std::vector` becomes too small. A similar approach was chosen by Lindow [Lin10]. During circle computation, the circle with a neighboring sphere σ_j is computed and then checked against all already present circles c_k for intersection. While iterating the already present circles, if a buried one is encountered, its index is then used to overwrite the corresponding buried circle with the current one, if it itself is to become part of the list as it is not buried. Note that it is possible that buried circles remain in the final circle list. The expensive circle intersection computation between c_j and c_k can be entirely skipped if the condition

$$\langle \vec{v}_j, \vec{n}_j \rangle < 0 \ \wedge \ \langle \vec{v}_k, \vec{n}_k \rangle < 0 \wedge \langle \vec{v}_j, \vec{v}_k \rangle < 0 \tag{4.2}$$

holds. If this condition is fulfilled, it does not matter if there is an intersection between the circles or not, c_j and c_k will never bury each other. An example can be found in Figure 4.2.

4.1.3 Contour Computation

While the second phase of the algorithm is more complex to implement, it requires less computation time than the circle computation. The implementation follows the approach described in Section 3.3.2. Again, arcs are written to memory managed by `std::vector`

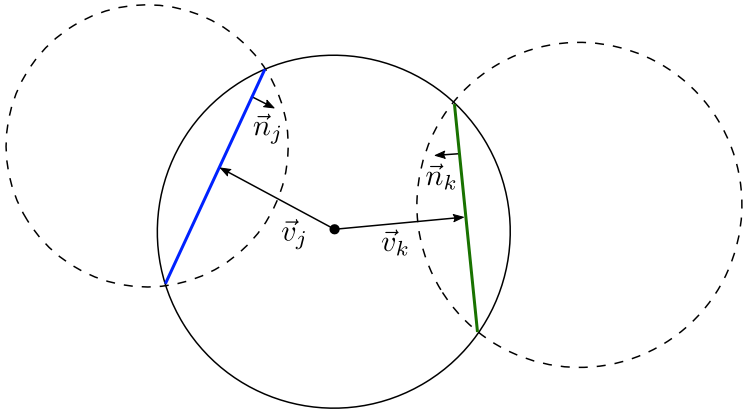

Figure 4.2: Example situation where the intersection computation between the blue and green circle can be skipped since Equation 4.2 holds.

Listing 4.3 The arc struct used in the implementation.

```
1    struct Arc {
2        int circleIdx;
3        vec3r s, e;
4        int psi, tau;
5        bool removed;
6    };
```

instances. In the `Arc` struct in Listing 4.3, if `psi` and `tau` are not negative, they refer to the previous and next arc. If both are negative, the arc consists of an entire closed circle. The corresponding circle is given by `circleIdx`. The `removed` flag indicates whether an arc is removed as it is covered completely. This part of the algorithm iterates over the computed circles as described in Section 3.3.2, while buried circles are skipped. The current circle is intersected with all circles that correspond to the current arcs. In case of an intersection, the arc has to be tracked to repair the contour as it is broken by the current circle. Listing 4.4 shows the `ModifiedArc` struct, containing this information, which is tracked in another `std::vector` instance. `ChangedArcVertex` tracks which end(s) of the arc were modified by intersection with the circle. This corresponds to the $\hat{A}_n$ and E_n sets in Section 3.3.2. After all current arcs were processed, the modified arcs have to be repaired. Recall that all modified arc endpoints lie on the currently processed circle. In order to reconnect the contour to a closed cycle, the endpoints have to be sorted. The usage of expensive trigonometric functions can be avoided by exploiting the arc endpoint positions

Listing 4.4 The modified arc struct used in the implementation.

```
1    struct ModifiedArc {
2        int index;
3        real pseudoAngle;
4        ChangedArcVertex changed;
5    };
```

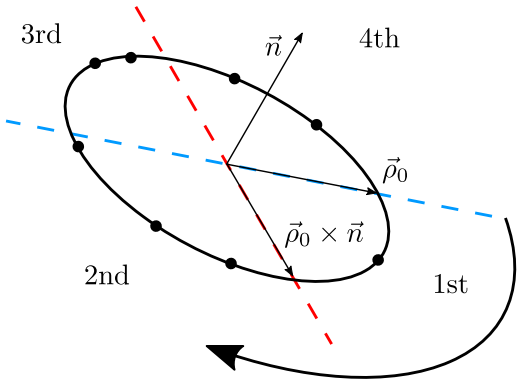

Figure 4.3: Depiction of the vectors used in the pseudoangle computation. The red dotted line depicts the plane defined by $\vec{\rho}_0$. Black points depict the points on the circle. The blue dotted line shows the plane defined by $\vec{\rho}_0 \times \vec{n}$. Both planes define four quadrants sorted in clockwise order viewed from above.

to compute a pseudoangle α. Note that no correct angle is required, i.e. to sort correctly, the quantity used must ensure

$$\alpha(x) < \alpha(x'), \tag{4.3}$$

where x and x' are arc endpoints, when x sits on the circle before x'. Such a pseudoangle α of a position $\vec{\rho}$ relative to an initial position $\vec{\rho}_0$ is computed from

$$\mu = \langle \vec{\rho}_0, \vec{\rho} \rangle, \tag{4.4}$$
$$\delta = \langle \vec{\rho}_0 \times \vec{n}, \vec{\rho} \rangle, \tag{4.5}$$
$$\lambda = \|\vec{\rho}_0 \times \vec{\rho}\|, \tag{4.6}$$

where $\vec{n}$ is the current circle's normal. Note that $\vec{\rho}$ and $\vec{\rho}_0$ are relative to the circle center. Geometrically, μ and δ divide the circle in four quadrants by their signs. The vector $\vec{\rho}_0$ defines a plane normal. Figure 4.3 shows a depiction. Therefore μ's sign decides which side $\vec{\rho}$ is located on. Similarly, $\vec{\rho}_0 \times \vec{n}$ defines a plane that contains $\vec{\rho}$ and is perpendicular to the first plane. The quantity δ describes $\vec{\rho}$ side on the plane. The third quantity λ can be rewritten to

$$\lambda = \underbrace{\|\vec{\rho}_0\| \|\vec{\rho}\|}_{\kappa} \sin\theta, \tag{4.7}$$

where θ is the angle between $\vec{\rho}_0$ and $\vec{\rho}$. Since $\vec{\rho}_0$ and $\vec{\rho}$ point to positions on a circle with radius R they have the same length, and do not change for any $\vec{\rho}$. Therefore κ can be seen as constant and $\kappa \equiv \langle \vec{\rho}_0, \vec{\rho}_0 \rangle \equiv R^2$. Finally, it holds that $\lambda \in [0, \kappa)$, and $\lambda \propto \sin\theta$ can now be seen as a quadrant-local pseudoangle. To obtain a circle wide angle α, Table 4.1 is used to correct λ. Now the angles describe a clockwise order when viewed against $\vec{n}$. To obtain clockwise order observed from the other side (i.e. viewed from the opposing sphere), all $\vec{\rho}$ can be ordered by decreasing angle, with $\vec{\rho}_0$ implicitly being the first element. Due to low number of such points to sort, simple selection sort is feasible to use.

Table 4.1: Correction of λ to use to obtain pseudo angle α. The quadrants are ordered clockwise.

$\mu > 0$	$\delta > 0$	Quadrant	$\alpha =$
T	T	1st	λ
T	F	4th	$4\kappa - \lambda$
F	T	2nd	$2\kappa - \lambda$
F	F	3r	$2\kappa + \lambda$

4.1.4 Memory Usage and Parallelization

While the approach can be implemented efficiently with separate `std::vector` instances for each sphere, this becomes problematic for large datasets, as for each sphere a `std::vector` has to be constructed and destroyed after usage implying overhead since the operating system has to be called for acquiring and releasing memory for every single sphere. Therefore a fixed number of `std::vector` instances is created, and subsequent circle and contour computations append to these vectors. In general, the sphere indices are divided into blocks. Each block contains the data of N' spheres where N' is computed from the total number of spheres N. The sphere with index i must finish appending items to its block before the sphere with index $i+1$ is allowed to append data. Each sphere is associated with a number of items, which is saved in a separate `std::vector`. For correct addressing, the index of the first item of each sphere is saved as well. Therefore each block contains one data vector and two data management vectors. The contour buildup algorithm is parallelized over the individual blocks. Since the spheres are divided into blocks, each thread computes the data of a block sequentially, which requires no explicit synchronization. This differs slightly from the approaches of Lindow et al. [Lin10] and Krone et al. [KGE11], where parallelization was applied over all spheres. Figure 4.4 shows a graphical representation of the blocked memory.

4.2 Vectorized Contour Buildup

Since the ISPC language is designed to allow porting of scalar to vectorized code with relative ease, the contour buildup implementation was directly ported as is. The parallelization was again applied to the spheres of the dataset. As ISPC does not offers container structures similar to `std::vector`, this functionality was implemented by arrays that are reallocated if their size becomes too small. However, straightforward porting of the contour buildup code is not as well behaved as expected, as can be seen in Section 5.1, which shows that the vectorized implementation's performance is worse than the scalar implementation. Performance decrease can be explained by several reasons. Firstly, the code is highly branching as different cases in the algorithm require completely different handling. This implies diverging code paths which do not allow simultaneous processing. The execution then becomes similar to execution of scalar code with added overhead by the vectorization. Next, the workload over spheres is varying depending on their amount of neighbors and

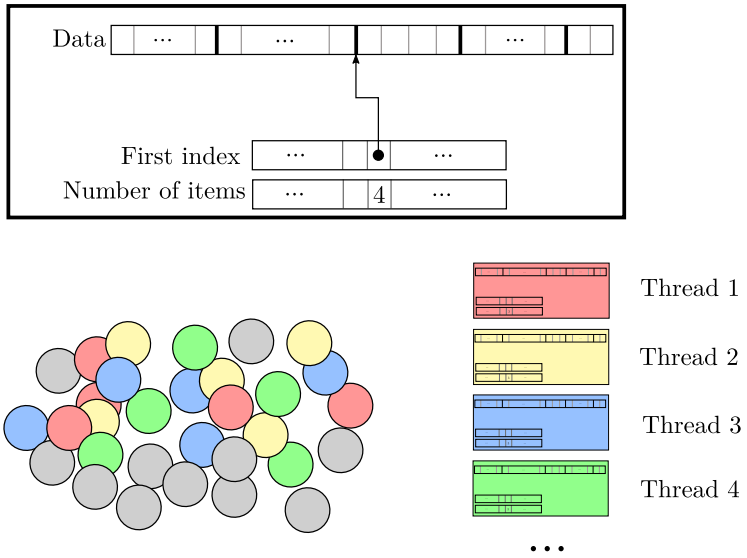

Figure 4.4: Depiction of the memory management approach used in the contour buildup implementation. Each sphere's data in the data array is indexed by the first index array. The number of associated items corresponds to the subsequent items that belong to the sphere. The dataset is divided in multiple blocks (colored boxes) that correspond to spheres in the dataset. Spheres belong to the block of the same color.

the computed contour. Consider the circle computation of a sphere with low number of neighbors and another sphere with high number of neighbors. If both spheres are computed simultaneously in a vectorized manner, the program instance handling the sphere with low number of neighbors is finished earlier and therefore stalls until the other sphere is processed. Finally, optimizations that allow skipping of expensive computations such as Equation 4.2 are less effective at best and introduce additional overhead at worst. If just a single program instance has to do the expensive computation, all other instances are stalling until the computation is finished. In the scalar implementation, the expensive computation is always skipped efficiently if possible. In general, differing `for`-loop lengths impose overhead. In some cases, the branching can be reduced by lookup tables. For example when arcs are modified, their endpoints are changed. Each case can be encoded as a binary number which in turn can be used to index into lookup tables. A modified arc a' either is assigned its old endpoint $\vec{s}(a)$, the intersection $\vec{x}_1$ or $\vec{x}_2$. This can be formulated as

$$\vec{s}(a') \leftarrow s_1\vec{s}(a) + s_2\vec{x}_1 + s_3\vec{x}_2 \tag{4.8}$$

by choosing the factors $s_1, s_2, s_3 \in \{0, 1\}$ appropriately. The factors s_1, s_2, s_3 are looked up from lookup tables depending on the case's binary number.

Partial vectorization of the algorithm was also tested. Specific parts of the algorithm could be vectorized. At this positions, ISPC functions are called while the rest of the code remains purely scalar. As there is no copying of data involved this could be beneficial even if the majority of the code is scalar. In this case, parallelization occurs over internal `for`-loops involved in the computation of the contour of a single sphere. Multiple places of

the algorithm were vectorization may be beneficial were identified. In the case of the circle computation two places were found: One possibility is vectorization of the computation of the intersection circles between extended spheres. Another possibility is vectorization over the computation of the intersection between all current circles and the current, fixed circle. In the case of contour computation, the intersection computation of the current circle with all arcs can be vectorized over said arcs. First all intersections between the circle and arcs are computed vectorized and written to (dynamic) memory. Computed are the intersection positions x_1 and x_2, as wells as the quantities describing the cases d_1, d_2 and d_3. Afterwards this result is read again and arcs are modified accordingly. In the case of the arc computation the detour over dynamic memory requires data to be stored and read one additional time, while in the scalar case all data necessary for computation stays on the stack and is never written somewhere else. All tested approaches proofed slower than the scalar implementation. In all cases, partial vectorization suffers from the fact that only a low number of items is processed in each loop (e.g. spheres have roughly 30 to 40 neighbors), this means that the overhead produced by vectorization may become significant enough to mitigate any gains achieved from parallel execution.

4.3 Trilateration Approach

During development, another potential approach for the computation of the solvent excluded surface was briefly examined. Recall that the contour of the extended spheres coincides with the probe's path as it rolls on the spheres, and that at each position where three contour arcs meet the probe sits locked in place by three spheres. Therefore, each triple crossing produced by the contour buildup algorithm is the intersection point of three extended spheres. All these triple crossings can be found by intersecting all possible triplets of spheres that are close enough together to produce up to two intersections per triplet. Finding the intersections resembles the problem of determining an unknown position from three known tracker locations and corresponding range estimates, which is known as *trilateration* [Fan86]. After all triplets were processed, all intersection points that are inside extended spheres are removed, as these are covered such that no probe can be located at this point. From the remaining points, immediately the spherical triangles of the SES can be derived. To avoid unnecessary intersection point computations, only triplets with sphere indices $i < j < k$ are examined, where i is the index of the first sphere, j is the index of the second sphere, and k is the index of the third sphere. Further, to find the triplets a grid similar to the one presented in Section 4.1.1 can be used for neighborhood computation. Still, this is an inherently cubic approach. To cull the buried triplet intersection points, again such a grid can be used to find possible spheres that contain the intersection point. This can be further accelerated by constructing a *fast reject grid*. Consider a grid dividing the space of the sphere dataset's bounding box. Then a grid cell can be marked if it is entirely contained in a extended sphere. After all spheres were processed, intersection points can be rejected in $O(1)$ if they fall in such a grid cell. However If their grid cell is unmarked, the neighborhood around the point must be searched for burying spheres.

Despite all these optimizations, just finding the intersections was slower than the entire contour buildup algorithm. Note that the toroidal patches are also yet to be determined, which is not trivial. Due to these circumstances, the approach was dropped in favor of the contour buildup algorithm. However, with further optimizations the approach might work well in a massively parallel environment such as GPUs.

4.4 Surface Rendering

For the implementation of the surface rendering in the OSPRay Framework, the individual surface elements, i.e. convex spherical patches, spherical triangles and toroidal patches have to be generated from the contour arcs. OSPRay offers the possibility of defining custom geometries that can be used such as any other geometry by the API [WJA+17]. Such custom entities must be part of a dynamic library that is loaded during runtime. The centerpiece of custom geometries are the callbacks for intersection, shading and bounding box computation. The entire surface is implemented in a single geometry, where each primitive has its own global index independent of its type. Data required for rendering is filled by the internal contour buildup algorithm or filled from external sources. In addition to the input sphere's radius and position, each sphere is additionally associated with an index that assigns a specific color, which is then interpolated over the surface during shading.

During rendering the primitives are read from individual arrays associated with each surface type. To index into the arrays, the primitive's global index is mapped to a local index indicating the n-th primitive of a specific type. Any cutting planes used in the following are of the form

$$P = (n_x, n_y, n_z, d)^\intercal, \tag{4.9}$$

where $\vec{n} = (n_x, n_y, n_z)^\intercal$ is the planes normal, and d the signed minimum distance to the origin. This corresponds to the Hesse-Normal form, however the signed d allows flipping the normal towards the origin if required. A side test is then done by

$$w = \langle \vec{n}, \vec{x} \rangle - d > 0, \tag{4.10}$$

which divides the sides into two half spaces. In the following, the side with $w > 0$ refers to the *positive halfspace* and $w < 0$ refers to the *negative halfspace*. For example, if both the normal $\vec{n}$ and d are negated, the plane's normal points from the plane towards the origin. All such cutting planes are written into a single array. Due to numerical errors, in the implementation the side test is done with a small threshold ϵ as

$$\langle \vec{n}, \vec{x} \rangle - d > \epsilon, \tag{4.11}$$

which allows neighboring surfaces to overlap slightly, reducing holes at patch boundaries.

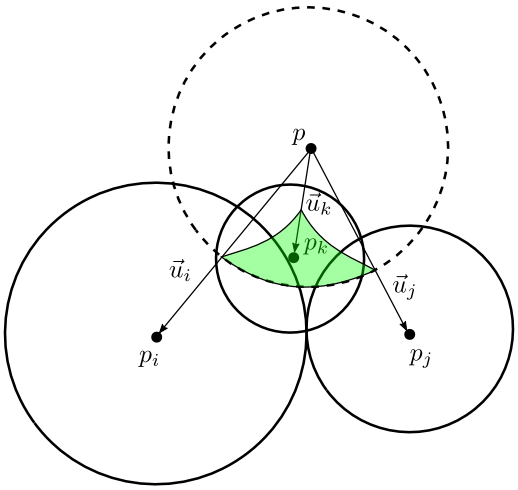

Figure 4.5: Depiction of the vectors used for computation of the cutting plane.

4.4.1 Spherical Triangles

Recall that spherical triangles appear when the probe is in contact with three spheres σ_i, σ_j and σ_k at once. The spherical triangle is produced by cutting a sphere with radius R_{probe} with planes and potential intersecting other probes resulting from neighboring spherical triangles. Each spherical triangle is associated with its probe position, three cutting plane indices, a neighborhood count and beginning index as well as the three contact spheres.

Cutting Geometry

To find intersection, first the ray is intersected with the probe sphere and afterwards the intersection point is potentially cut away by the three cutting planes. The probe's contact points lie on the ray between the probe center p and the corresponding sphere centers. Therefore each plane's normal can be computed by taking the normalized cross product between all combination of

$$\vec{u}_i = p_i - p, \qquad \vec{u}_j = p_j - p, \qquad \vec{u}_k = p_k - p. \tag{4.12}$$

See Figure 4.5 for a depiction. Let the remaining part of the probe sphere by defined by the intersection of the positive halfspaces defined by all three planes. The sphere position that is not used in computing the plane is used as a *witness* to flip the normal and d if necessary, as this point must be located in the positive halfspace, e.g. if

$$\langle \vec{u}_i \times \vec{u}_j, \vec{u}_k \rangle - d < 0. \tag{4.13}$$

The resulting cutting plane is local to the spherical triangle's probe position. Additionally, the singularities must be handled, i.e. intersecting spherical triangles must be mutually cut away. This is achieved by finding all possible intersecting probe positions for each spherical patch, and then using the probe spheres as cutting geometry. To dampen the quadratic effort to find the intersecting probes, a grid similar to the neighborhood computation is

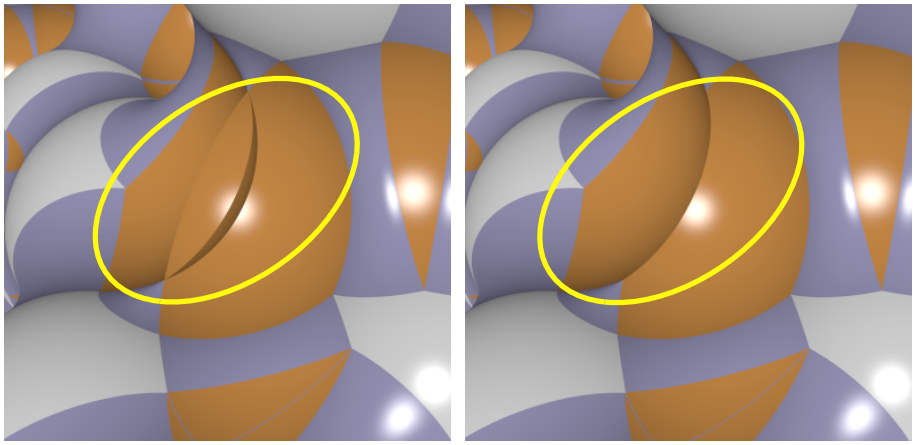

Figure 4.6: Singularity occurring between two spherical triangles (left), that is subsequently removed (right).

used with grid cell length $\max(1, 2R_{\text{probe}})$. The minimum grid cell length is bounded as R_{probe} can be chosen arbitrarily small. Figure 4.6 shows an example where a singularity occurs and is removed.

4.4.2 Toroidal Patches

The toroidal patch is generated from an entire torus. Cutting geometry consisting of a sphere and two planes is used to cut the torus. Additionally, numerical inaccuracies during the intersection computation have to be considered.

Cutting Geometry

The *visibility sphere* [KBE09] is used for clipping a torus located at c between σ_i an σ_j. Everything outside the sphere is clipped, leaving the toroidal patch behind. The visibility sphere's position p_{vs} and radius R_{vs} is computed as

$$\vec{x} = \frac{p - p_i}{\|p - p_i\|} R_i, \tag{4.14}$$

$$\vec{d} = \frac{\|p - p_i\|}{\|p - p_j\| + \|p - p_i\|} (p_j - p_i) \tag{4.15}$$

$$p_{\text{vs}} = \vec{d} + p_i - p, \tag{4.16}$$

$$R_{\text{vs}} = \|\vec{x} - \vec{d}\|. \tag{4.17}$$

Here $\vec{d}$ is the position vector of the visibility sphere relative to p_i. Figure 4.7 shows an example. For opaque SES rendering cutting with the visibility sphere is all that is required to obtain the toroidal patch. However, for transparent rendering the part of the torus below the surface has to be cut additionally. Recall that a toroidal patch is generated from an arc a that is local to σ_i. The arc endpoints $\vec{s}(a)$ and $\vec{e}(a)$ are used to compute the two

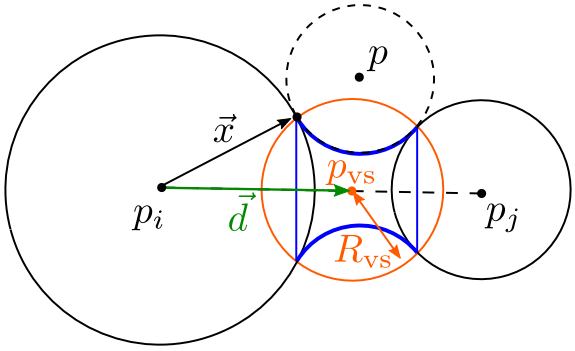

Figure 4.7: Example visibility sphere (orange) for a toroidal patch (blue).

cutting planes, with the normal vectors pointing outwards (in other words, away from the patch). The angle of the arc determines how the cutting planes are to be used. If the angle is $< \pi$, a point is not cut away if it is located in the intersection of both negative halfspaces defined by the planes. Otherwise the point is not cut away if it is located in the union of negative halfspaces. Jurcik et al. compared this to AND and OR operations [JPSK16]. First every point is made local to the toroidal patch center c:

$$\vec{o}_i = p_i - c, \tag{4.18}$$
$$\vec{o}_j = p_j - c, \tag{4.19}$$
$$\vec{q}_s = \vec{s}(a) + p_i - c, \tag{4.20}$$
$$\vec{q}_e = \vec{e}(a) + p_i - c, \tag{4.21}$$

Then the normals are computed by taking the cross product of the sphere positions $\vec{o}_i$ and $\vec{o}_j$ both local to one of the arc endpoints, as

$$\vec{n}_1 = \frac{(\vec{o}_i - \vec{q}_s) \times (\vec{o}_j - \vec{q}_s)}{\|(\vec{o}_i - \vec{q}_s) \times (\vec{o}_j - \vec{q}_s)\|}, \tag{4.22}$$
$$\vec{n}_2 = \frac{(\vec{o}_i - \vec{q}_e) \times (\vec{o}_j - \vec{q}_e)}{\|(\vec{o}_i - \vec{q}_e) \times (\vec{o}_j - \vec{q}_e)\|}, \tag{4.23}$$

and then corrected by considering the other arc endpoint. The sign of the quantities

$$d_1 = \langle \vec{q}_e - \vec{q}_s, \vec{n}_1 \rangle,$$
$$d_2 = \langle \vec{q}_s - \vec{q}_e, \vec{n}_2 \rangle,$$

describe the side on which the other arc endpoint is located, relative to the normal. If the arc angle is $< \pi$, the other arc endpoint must be on the negative side (side the normal points away from) i.e. $d_{1,2} < 0$, otherwise the normal must be negated. Similarly, if the angle is $> \pi$, the other arc endpoint must dip below the plane and be on the positive side, in order to allow cutting as described above. Figure 4.8 shows a depiction.

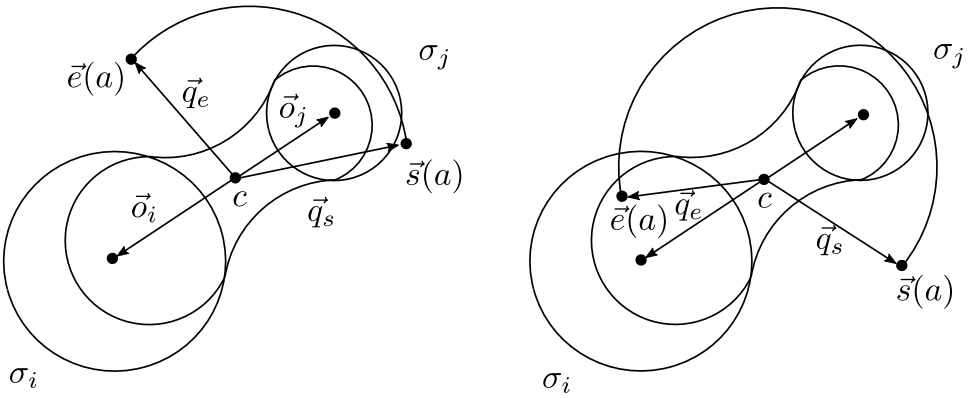

Figure 4.8: Depiction of the vectors used to compute the cutting planes of the toroidal patches. The left side shows the case for angle $< \pi$, the right side shows the case for angle $> \pi$.

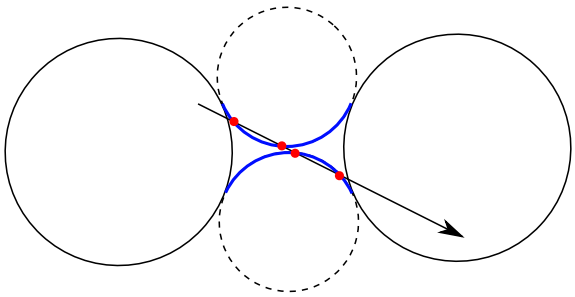

Figure 4.9: Example situation where a ray intersects a toroidal patch (blue) in four positions (red).

Intersection Computation

Three different intersection schemes were tried, the analytical solving of the ray-torus quartic based on the stabilized Ferrari algorithm (Section 2.2.1), as well as two iterative methods based on the Bairstow method (Section 2.2.2) and sphere tracing (Section 2.1.2). The Ferrari algorithm was also used by Krone et al. [KBE09]. Sphere tracing was used by Lindow et al. [LBPH10]. Note that all four intersections are required, in contrast to opaque rendering, as shown in Figure 4.9. Krone et al. [KBE09] used only two intersections, as this is sufficient if no transparency is rendered and the ray origin is located outside of the molecule's convex hull.

In the implementation of the analytic solution of the ray-torus intersection problem, the Ferrari code provided by Herbison-Evans [Her95] was ported to ISPC code. First the ray is transformed such that the torus is centered at the origin and parallel to the xy-plane. To increase numerical precision, the viewing ray's origin is translated close to the torus, by using the distance function [Har96]

$$d(\vec{x} = (x, y, z)^\intercal) = \|(\|(x, y)^\intercal\|) - R, z)^\intercal\| - r \tag{4.24}$$

to translate the ray forward. Moving the ray closer to the torus was also proposed by Krone et al. [KBE09]. However still in some rare cases the stabilized Ferrari algorithm will fail and miss intersections as shown in Section 5.2. These numerical problems were also reported by Jurcik et al. [JPSK16].

Therefore iterative schemes were tried in an attempt to reduce the artifacts. For torus intersection, de Toledo et al. [TLP07] found the Newton-Raphson iteration to work best, however only the first intersect was needed. The general "W"-shape of the quartic allows rapid and secure convergence if the starting t is chosen as being smaller than all possible intersections t_1, t_2, t_3, t_4. Such a initial value is simple to obtain by first intersecting with bounding geometry, e.g. a sphere. The same holds for the last intersection. However, iterating the internal intersection points t_2 and t_3 becomes difficult as the inner intersections t_2 and t_3 can be distributed arbitrarily. Any starting position chosen in (t_1, t_4) might lead to Newton-Raphson iteration converging to t_1 of t_4 again. Therefore the Bairstow method was tried. This approach seemed promising as applying Bairstow's method to a quartic immediately gives all four roots after the iteration converges. There is however the problem of numerical instability introduced by polynomial deflation. Section 5.2 shows the artifacts of this method.

The second iterative approach based on sphere tracing was implemented with the torus distance function given in Equation 4.24. While the standard sphere tracing method works reasonably well for finding the first intersection of whole tori [TLP07], and toroidal patches of the SES [LBPH10], transparent rendering requires some extra measures. Recall that an intersection is accepted if the distance to the surface drops below a threshold ϵ. Let the *ϵ-region* be the region where an intersection is accepted. In principle it will be impossible to escape this region as one converges towards the surface until it is infinitesimally close. Doing a single constant step to escape the ϵ-region does not work in situation where the ray is roughly parallel to the surface, as the ray will then possible stay in the ϵ-region. Therefore, a two phase approach is chosen: First, sphere tracing is iterated until a ϵ-region is encountered, where a intersection is registered. Afterwards, iteration continues along the ray with constant step length until the region is left again. Then, standard sphere tracing continues along the ray until the next intersection is encountered where this procedure repeats. Since the visibility sphere encloses the toroidal patch, it is first intersected to find a starting point for sphere tracing. As soon as the visibility sphere is left again, the iteration is stopped, as no valid intersection can then be found anymore.

4.4.3 Convex Spherical Patches

If no transparency is required and the viewer stays outside the surface, all spheres of the dataset can be rendered directly without any further computation of cutting geometry. The part below the SES will never be visible and does not have to be clipped in this case, making it the simplest primitive to render. However, transparency rendering requires removal of buried spheres and cutting of internal parts. The approach shown by Kauker et al. [KKP+13] is not well suited for an implementation in OSPRay. It would require tracing the ray onwards through the entire scene after intersection to determine if the current intersection point is to be removed. Implementation would be highly inefficient for all rendering methods were the viewing ray does not pass through the surface in a

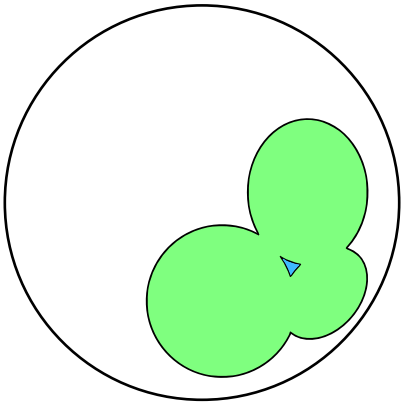

Figure 4.10: Example depiction where cutting with the visibility spheres of connected toroidal patches does not entirely remove spherical remains. The cut away surface is colored in green, while the remaining splinter is colored blue.

straight line, such as path tracing. Additionally a custom renderer would be required which allows the CSG operation proposed by Kauker et al., making transparency rendering with OSPRay's standard renderers impossible. Therefore a localized clipping approach similar to the one proposed by Jurcik et al. [JPSK16] is required. Two clipping approaches were developed, which will be described in the following.

Ray-Triangle Based Cutting

Recall that a convex spherical patch is part of a sphere σ_i of the input dataset of spheres. With reasonable probe radius and a realistic input dataset, most of the time the SES lets only a small part of the original sphere exposed. However in general, the connected toroidal patches may carve away almost arbitrary shapes on the sphere, such as a large sphere surrounded by many small ones. While the visibility sphere bounds the toroidal patch, it can also be used to cut away large parts of the spherical remains below the surface, as the visibility sphere intersected with the sphere results in a intersection circle exactly where the corresponding toroidal patch is connected. In many cases, cutting with the visibility sphere is sufficient. However, sometimes spherical parts remain if the visibility sphere do not cover the internal region completely, as seen in Figure 4.10. In this case additional clipping geometry has to be generated. The proposed method is based on triangle intersection and inside/outside testing. Consider an arc a and its arc endpoints $\vec{s}(a)$ and $\vec{e}(a)$. Then these two points together with the sphere center c (locally the origin) can be used to generate a triangle. As the contour is closed, generating all triangles of all arcs will result in a structure that is closed towards the center of the sphere, as all triangles meet there, and no holes are generated. Note however, that the generated structure's top is still open. Each triangle is constructed such that the normal vector points inwards. Intuitively, the triangles build a ravine-like structure that encloses the spherical remains. Any ray that is cast from a valid surface point will have its closest intersection, if any, with a triangle that is facing away from the point. Additionally, any ray that is cast from spherical remains will have its closest intersection with a triangle facing towards the point, as long as the ray does not

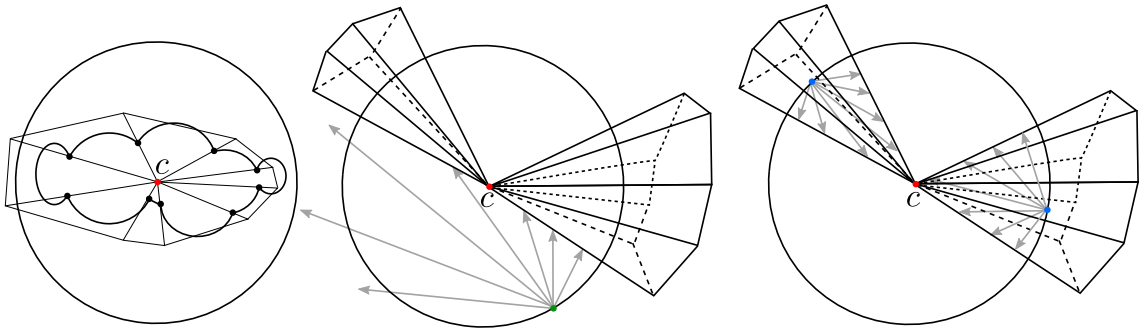

Figure 4.11: Left: Frontal view of the triangles. The red point c is the center of the sphere, where all triangles meet. Black points depict the arc ends projected on the sphere. Triangles are generated by connecting c and an arc's start and end point. The center and right figure shows example rays cast from surface points. The green point is not cut away, since the closest intersection of all rays with a triangle hits on the outward facing side if any intersection is possible. In contrast, the blue points are cut away since all rays with roughly downward direction first intersect a triangle on the inward side.

leave through the opening on top. Therefore it is crucial to select a ray with reasonable direction, i.e. roughly downwards towards the center c. To obtain such a direction, one may select the barycenter of one of the triangles and aim the ray towards it. This will guarantee at least one intersection. Figure 4.11 shows a depictions of the approach. To sum up, if a surface points survives the cutting test with the visibility spheres it is additionally tested for cutting with the approach described above.

Circle Plane Based Cutting

This approach is simpler than the one described in Section 4.4.3. Consider the intersection circles computed in Section 3.3.2. The result of this computation is used to compute cutting planes that remove all spherical parts below the surface. Consider the sphere σ_i and its neighbor σ_j that produces a intersection circle c by their extended radii. In absence of other spheres, the SES will be compromised of two convex spherical patches and one toroidal patch. The contact boundaries where spherical patch and toroidal patch meet are circles. From these directly the cutting plane is derived to remove the spherical remains of σ_i (and vice versa for σ_j). All circles are used for computation of such cutting planes, as adding more spheres in the vicinity of σ_i will never lead to its previously cut surface becoming exposed again, as σ_i only ever loses possible contact area with the probe as more spheres are added in its neighborhood. Only those circles that are not covered are required for cutting, as the surface cut away by cutting planes computed from covered circles is included by the planes computed from the circles covering them. Every uncovered circle c_j corresponding to σ_j is computed by projecting the tangent point where σ_i and probe

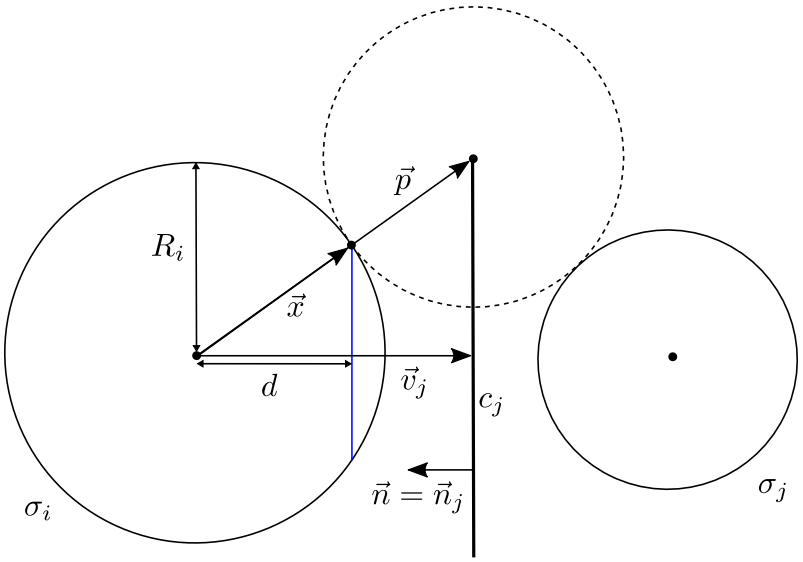

Figure 4.12: The vectors and measures used for generating a single cutting plane from a circle c_j, depicted in blue.

sphere meet onto the circle normal. The cutting plane normal $\vec{n}$ for sphere σ_i is just the circle normal $\vec{n}_j$, while the distance d is computed as

$$\vec{x} = R_i \frac{\vec{p}}{\|\vec{p}\|}$$
$$d = \langle n_j, x \rangle$$

where $\vec{p}$ is a probe position vector on c_j. $\vec{p}$ can be found by choosing a vector $\vec{u}$ perpendicular to the circle normal n_j and then computing

$$\vec{p} = \vec{v}_j + r(c_j) \frac{\vec{n}}{\|\vec{n}\|} \tag{4.25}$$

where $\vec{v}_j$ is the circle's center and $r(c_j)$ the circle radius. Figure 4.12 shows a depiction of the used vectors.

In the final implementation, this approach was used in favor of the triangle based method described in Section 4.4.3, as it suffers from numerical issues in the rare cases where triangles become very small.

4.4.4 Color Interpolation

In the following sections the interpolation of color on the surface is described. The approach is somewhat arbitrary, as there is no specific physical motivation behind it. Interpolated color only conveys spatial closeness of the spheres to the surface. Note that the convex spherical patches are rendered directly without interpolation by rendering the sphere's assigned color. Figure 4.13 shows an example of the surface color computed by the interpolation.

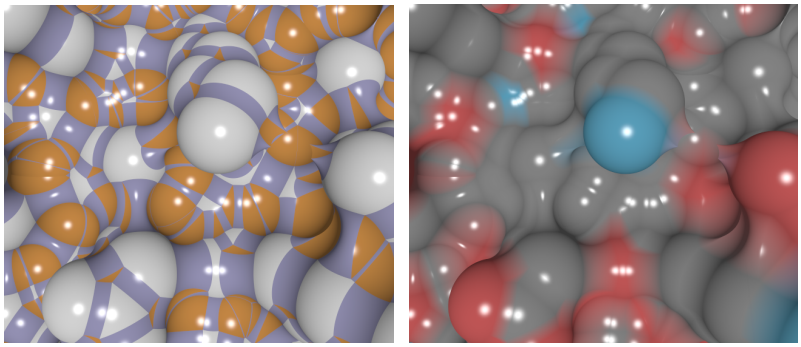

Figure 4.13: Color interpolation example. Left: The SES's individual surface patches are shown, where convex spherical patches are white, spherical triangles are green and toroidal patches are red. Right: Color interpolation computed from the colors of the convex spherical patches.

4.4.5 Spherical Triangles

The color interpolation is done by ray-triangle intersection similar to the implementation by Krone et al. [GKM+15; KBE09] in MegaMol. Consider the three sphere positions p_i, p_j and p_k the spherical triangle's probe is in contact with, which span a triangle. The ray

$$\vec{r}(t) = p + t\frac{x-p}{\|x-p\|}, \tag{4.26}$$

where x is the current intersection point on the spherical triangle is intersected with the spanned triangle, which yields three barycentric coordinates α, β, γ. These are directly used to interpolate the spherical triangle's color from the contact sphere's colors, as the barycentric coordinates directly correspond to the triangle's corner points and therefore the sphere positions. Figure 4.14 shows an example. As the triangle is embedded in the cutting geometry, at the spherical triangle boundaries one of the coordinates will vanish and interpolation will become linear between the two remaining spheres.

4.4.6 Toroidal Patches

Any color interpolation scheme for toroidal patches located between σ_i and σ_j must produce the same weights at the border to neighboring spherical triangles as in Section 4.4.5, in order to be at least C^0 smooth. A geometric approach was chosen to achieve this. First the probe position of the intersection is computed, i.e. the probe position that is located in the plane spanned by the intersection position $\vec{x}$, p_i and p_j which is the position where the probe is in contact with $\vec{x}$. First the auxiliary vectors $\vec{f}$ and $\vec{d}$ are computed as

$$\vec{f} = (p_j - p_i) \times (x - p_i), \qquad \vec{d} = \frac{f \times (p_j - p_i)}{\|f \times (p_j - p_i)\|}. \tag{4.27}$$

Hereby $\vec{d}$ can be interpreted as the direction from the torus center c towards the probe position p (local to p_i), therefore

$$p = c + R\vec{d} - p_i \tag{4.28}$$

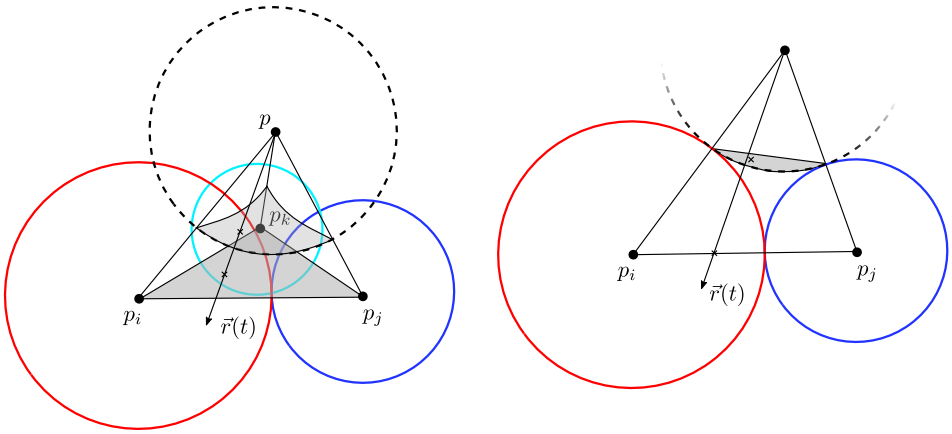

Figure 4.14: Depictions of the triangle generated from the three involved sphere positions p_i, p_j and p_k of a spherical triangle. Left: The ray $\vec{r}(t)$ is send from the probe center p towards the intersection point x, and intersected with the triangle to obtain barycentric coordinates acting as color weights. Right: Side view of the generated triangle.

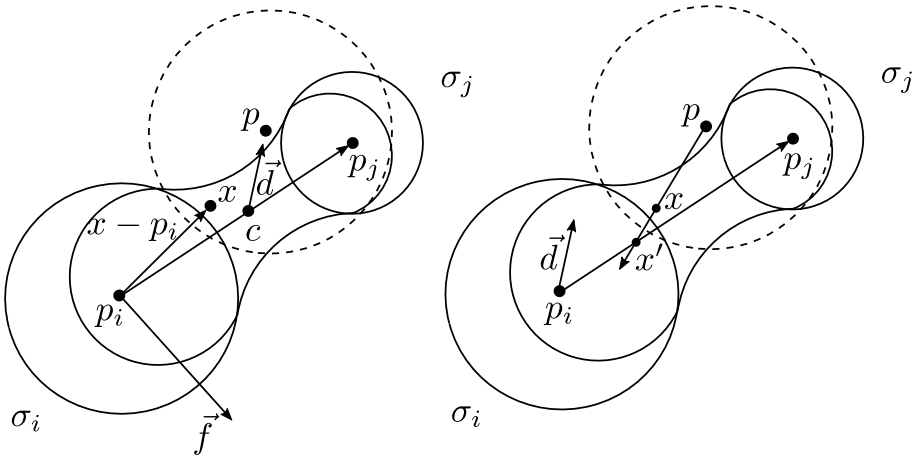

Figure 4.15: Example depiction of color interpolation on the toroidal patch. From the vector $\vec{d}$ the plane through the origin is constructed and intersected to obtain $\vec{x}'$, from which the interpolation weight is obtained.

where R is the torus major radius. Now a ray is cast from the probe position p towards the plane located at the origin with normal $\vec{d}$. This intersection $\vec{x}'$ will be located on $p_j - p_i$, as all involved positions are in the same plane. The interpolation weight w is then computed as

$$v = \|p_j - p_i\|, \qquad w = \frac{\langle \vec{x}', (p_j - p_i)/v \rangle}{v} \tag{4.29}$$

where $w \in [0, 1)$. The weight w is the projection of $\vec{x}'$ onto normalized $p_j - p_i$, which is then itself normalized afterwards. Figure 4.15 shows an example. Note that on the border to a neighboring connected spherical triangle, $p + p_i$ coincides with the spherical triangle's probe

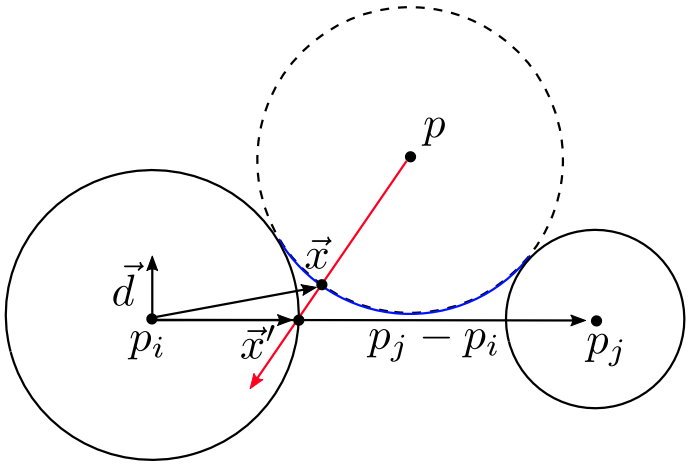

Figure 4.16: Side view example of the color interpolation. A ray(red) is cast towards the intersection with the toroidal patch (blue). The interpolation weight is obtained from the ray intersection at $\vec{x}'$.

position. Therefore w corresponds to the barycentric coordinates, as $p_j - p_i$ corresponds to the triangle edge. Figure 4.16 shows a side view example. Compare with Figure 4.14 to observe that w is computed equivalently to the barycentric weight at the border of a spherical triangle, leading to C^0 smoothness.

4.4.7 Bounding Boxes

Embree's bounding volume hierarchy requires tight fitting axis-aligned bounding boxes for geometry. In the following sections the approaches taken to obtain bounding boxes for each primitive type are described. The approaches compute a set of points that is constructed in such a way that its convex hull contains the corresponding primitive. From these points the axis-aligned bounding box which contains all those points is computed.

Convex Spherical Patches

As the convex spherical patch is part of an entire sphere σ_i of the dataset, a simple bounding box can be obtained by fitting the sphere into a cuboid. However this bounding box is wasteful if large parts of the sphere is below the SES. Since these planes obtained in Section 4.4.3 cut away half spaces, they can be used to cut the simple bounding box. The bounding box is divided into three pairs of opposing faces: Left-right, lower-upper and back-front. Since all bounding boxes are axis aligned, in order for the cutting plane to reduced the box side, the plane must be located between one of the pairs without intersecting the faces. If this is the case, the new bounding box is found by casting rays from the corner points a, b, c, d that are not cut away by the plane towards the opposing face's corner points and intersecting the ray with the cutting plane, which yields four new

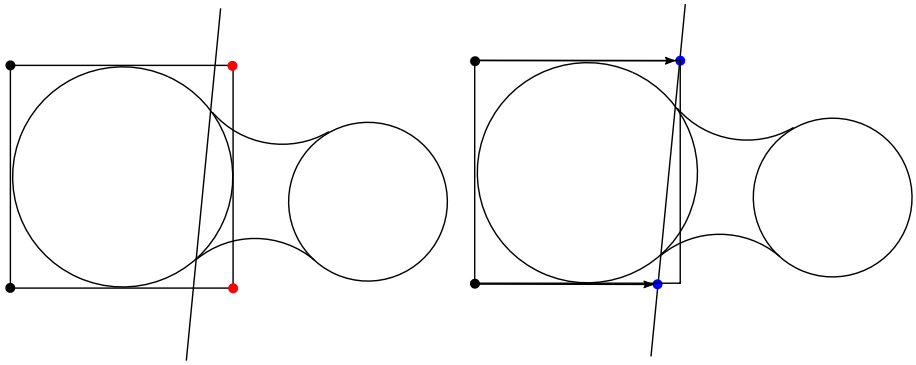

Figure 4.17: Depiction of the cutting of the convex spherical patch's bounding box. The cutting plane cuts away the red points of the bounding box. By intersecting the plane the blue points are found. From these a smaller bounding box is found.

points a', b', c', d' from the intersections. The new box is then constructed by creating an axis aligned bounding box for a, b, c, d and a', b', c', d'. Figure 4.17 depicts the approach.

Spherical Triangles

The simplest kind of bounding box can be derived from the spherical triangle's probe sphere. However depending on the rotational configuration of the spherical triangle, at least half of the bounding box is wasteful, as the spherical triangle at most covers the hemisphere of a probe sphere. An oriented bounding box for the spherical triangle can be computed as follows: First the plane defined by the three tangent points where the probe touches the spheres is constructed. On this plane, four points are arranged in a rectangular way around the intersection circle. These four points are translated by the height of the spherical triangle, yielding another four points. The axis aligned bounding box is constructed from these eight points. This approach is similar to the one presented by Lindow [Lin10]. Figure 4.18 shows an example bounding box.

This approach may lead to bounding boxes that may be even larger than the simple bounding box derived from the probe sphere. However both bounding boxes are valid in the sense that they completely cover the geometry. Both bounding boxes are additionally intersected to reduce the bounding box's size which yields a valid, but smaller bounding box.

Toroidal Patches

The visibility sphere can be used to obtain a reasonably good bounding box of the toroidal patch. In this case the two cutting planes are not considered. The following approach involves these. In case the arc a that corresponds to the toroidal patch's angle is $> \pi$, the bounding box is computed as follows: First the intersection circle between visibility sphere

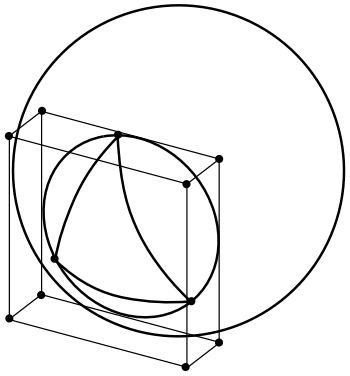

Figure 4.18: Depiction of the spherical triangles' oriented bounding box. The circle results from the intersection of the plane containing the corresponding sphere's tangent points with the probe sphere. The axis aligned bounding box is obtained from the corner points.

and both connected spheres is computed. The circles each coincides with the connection of toroidal patch and the connected spheres. Next the four points where the circle connects with the neighboring spherical triangles are computed from the arcs ends $\vec{s}(a)$ and $\vec{e}(a)$, called *patch corners* in the following. The halfway vectors between each of the point pairs is computed. From these, a rectangle is traced out. These rectangles each contain the arc that corresponds to the connection where toroidal patch and the corresponding sphere meet. Now from all eight corner points the bounding box is derived. See Figure 4.19 for a depiction. If the arc angle is $< \pi$, again two rectangles are traced out. However this is not sufficient, as it is possible that the toroidal patch bends outside of bounding geometry computed from these points. The probe position located at half the covered angle produces the lowest point in the bend that is not covered yet. This position is obtained from the arc ends $\vec{s}(a)$ and $\vec{e}(a)$ by computing their halfway vector and using it as a direction from the torus center. Additionally it is possible that the boundaries to neighboring spherical triangles contain the lowest point, which can be obtained by constructing vector from the torus center towards $\vec{s}(a)$ and $\vec{e}(a)$. Then the plane containing the patch corners is computed, and the distance of the plane to the lowest point is used to translate the patch corners in direction of the bend, enclosing the toroidal patch. Figure 4.19 shows an example. Again the simple bounding box is intersected with the complex one to possibly obtain a potentially better final bounding box.

4.5 Cavity Visualization

The surface cannot only be rendered in the classical sense by using opaque or transparent materials, but also be used to implement a real-time variant of AOOM. As described in Section 3.2, AO weights are mapped to opacity weights, producing images where cavities that appear shadowed according to the ambient occlusion weight are less transparent than exposed areas on the surface. However, the approach by Borland [Bor11] uses precomputed ambient occlusion values on a triangulated representation of the SES. By using real time

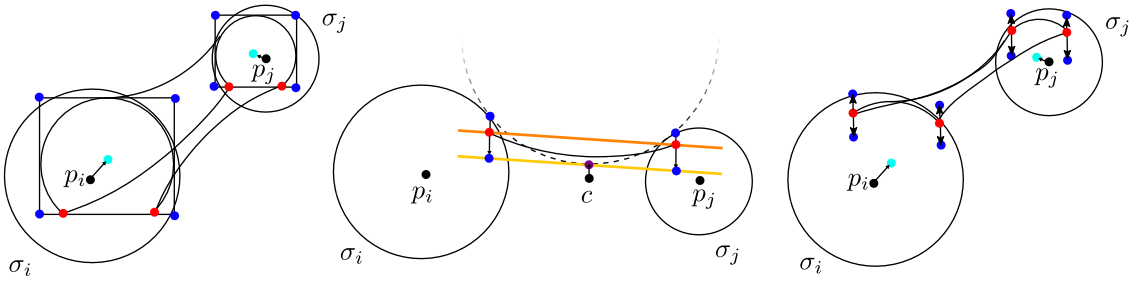

Figure 4.19: Depiction of the points used to construct the toroidal patch's bounding box. The left figure shows the case for angle $< \pi$, the right and center figures shows the case with angle $> \pi$. Circle centers are depicted in bright blue. The red points depict the patch corners where toroidal patch, sphere and neighboring spherical triangles meet. From these the blue points are obtained, which are used to obtain a axis-aligned bounding box. The center figure shows the plane constructed from the corner points (orange plane) which is shifted to contain the the point of the probe that produces the lowest point in the bend (purple).

ambient occlusion, the AO term is computed on the fly at each intersection and directly mapped to opacity. Accumulation, i.e. averaging of samples at pixels should be used to obtain a noiseless image after some time. At each intersection p, the AO term O_p is estimated by Monte Carlo sampling [PH10] the surrounding hemisphere. The used AO visibility function is

$$V_p(\vec{\omega}) = \min\left(1, \max\left(0, \chi(\vec{\omega})^{\rho}\right)\right) \tag{4.30}$$

where

$$\chi(\vec{\omega}) = \frac{\left(1 - \frac{D_p(\vec{\omega})}{D_{\max}}\right)}{\tau}, \tag{4.31}$$

which is inspired by the weighting scheme used by Borland [Bor11]. The AO sample's distance is denoted with $D_p(\vec{\omega})$, while the maximum distance is denoted with $D_{\max}$. Intuitively, $\chi(\vec{\omega})$ returns high values for close surfaces. This leads to small cavities always being largely occluded, regardless of $D_{\max}$. If larger cavities are of interest, the function can be chosen as

$$\chi(\vec{\omega}) = \frac{\left(\frac{D_p(\vec{\omega})}{D_{\max}}\right)}{\tau}, \tag{4.32}$$

which gives higher values for surfaces further away from p. Similar to the original AOOM method, $0 < \tau \leq 1$ allows faster saturation of the visibility function, while ρ allows nonlinear shaping. The UNDER operator shown in Section 3.3.4 should be used to correctly blend the colors at each intersection. Additionally, the AO term should be used for coloring, e.g. the color is computed as

$$C_{\text{src}} = \beta O_p C_{\text{blend}} + (1 - \beta O_p) C_{\text{surface}}, \tag{4.33}$$

where C_{blend} is the color indicating a cavity and β controls the influence of this color (which are both fixed), and C_{surface} is the surface color produced by interpolation. Now when the UNDER operator is used, the AO term will appear squared in the resulting equation as A_{src}

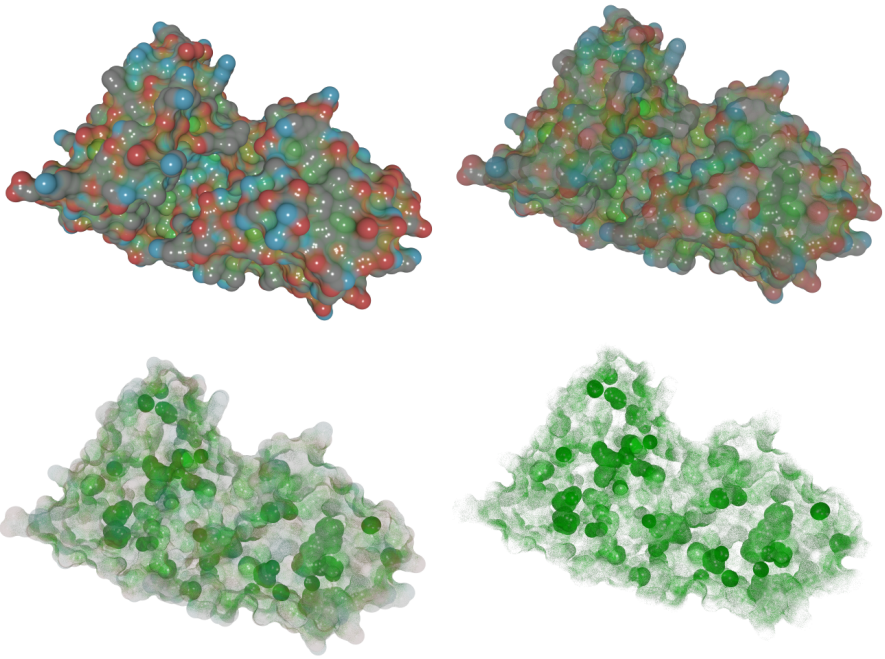

Figure 4.20: From left to right, top to bottom: The AOOM variant using Equation 4.31 with blending weight $\beta = 1$ for pure green and general surface opacity $\alpha = 1$, $\alpha = 0.5$, $\alpha = 0.1$ and $\alpha = 0$. The visualized molecule is *1vis*.

also depends on O_p, which is a non linear operator that prohibits usage of accumulation by averaging. Resulting images will otherwise be biased. Therefore the approach was modified to obtain an unbiased result under accumulation. During rendering, at each intersection blending is done twice by assuming that two infinitesimally close surfaces are intersected by the viewing ray at the same point. The first surface is colored according to

$$C_{\text{src}} = (1-\beta)C_{\text{surface}} + \beta C_{\text{blend}}, \qquad A_{\text{src}} = O_p \tag{4.34}$$

where C_{blend} is computed from the cavity indicating color and β again controls the influence of C_{blend}. The second surface is just colored according to the surface. Now the blending is then done as

$$C_{\text{dst}} \leftarrow A_{\text{src}} A_{\text{dst}} C_{\text{src}} + C_{\text{dst}}, \tag{4.35}$$
$$A_{\text{dst}} \leftarrow (1 - A_{\text{src}}) A_{\text{dst}}, \tag{4.36}$$
$$C_{\text{dst}} \leftarrow \alpha A_{\text{dst}} C_{\text{surface}} + C_{\text{dst}}, \tag{4.37}$$
$$A_{\text{dst}} \leftarrow (1 - \alpha) A_{\text{dst}}, \tag{4.38}$$

where α controls the general opacity of the surface. With $\alpha = 1$ and $\beta = 1$, the approach mimics ambient occlusion, where occluded areas are colored according to C_{blend}. Figure 4.20 and Figure 4.21 show the influence of changing these parameters. Unbiased accumulation is shown in Figure 4.22. The method is unbiased, as accumulated renderings with different numbers of AO samples per intersection still converge to the same image.

Figure 4.21: From left to right: The AOOM variant using Equation 4.31 with opacity $\alpha = 0.1$ for pure green and blending weights $\beta = 0$, $\beta = 0.5$ and $\beta = 1$. The visualized molecule is *1af6*.

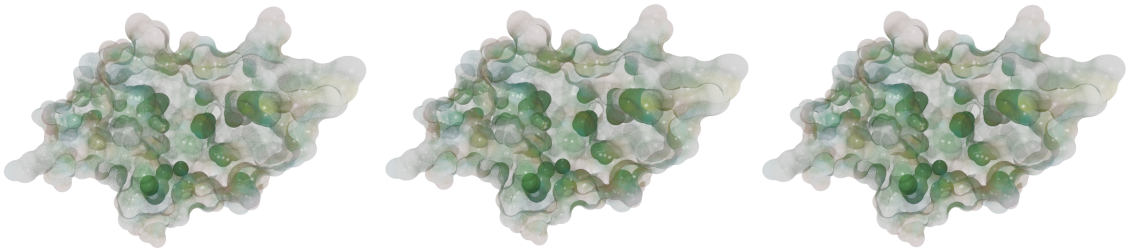

Figure 4.22: Accumulated renderings done with the AOOM variant implemented in this work. The visualized molecule is *1ogz*. From left to right: 1 AO sample per intersection, 5 AO samples per intersection and 10 AO samples per intersection. Convergence to the same result independent of sample count shows that the method is unbiased.

4.6 MegaMol Integration

The implemented OSPRay Geometry was also made available for usage in MegaMol. Two specific modules were developed for this: One is a renderer that allows switching between transparency rendering and the ambient occlusion opacity mapping implementation, called *OSPRaySESRenderer*. The other is a generic SES geometry that is used like the other OSPRay geometries, called *OSPRaySESGeometry*. This allows usage of all renderers and materials provided by OSPRay with the SES. See Figure 4.23 and Figure 4.24 for their MegaMol graphs.

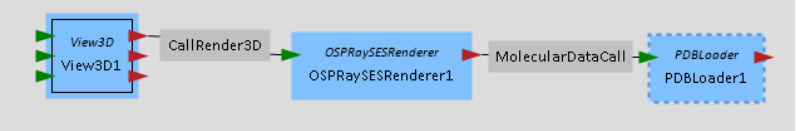

Figure 4.23: MegaMol graph for the renderer OSPRaySESRenderer implementing transparency and AOOM rendering. Image created with MegaMol's configurator tool.

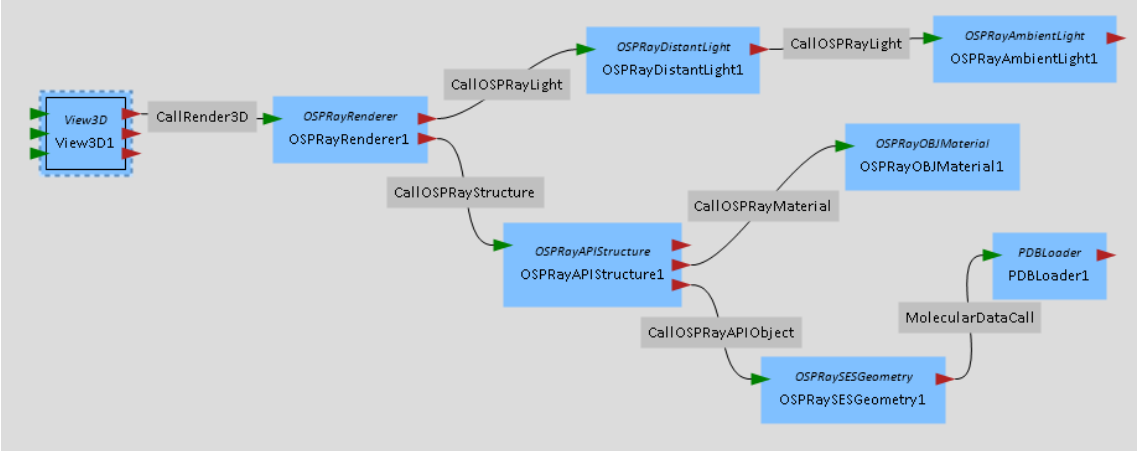

Figure 4.24: MegaMol graph for the geometry OSPRaySESGeometry, which wraps the OSPRay surface geometry implementation described in this work. Image created with MegaMol's configurator tool.

5 Results and Discussion

The following sections contain the results obtained from the implementation. Both the implementations of the contour buildup and the surface rendering were tested. For the contour buildup, the performance of the developed variants are discussed. The surface rendering concerns both quality and performance. Table 5.1 shows the number of atoms of each dataset. All tests use real world datasets from the RCSB protein databank[1]. The molecules *1vis*, *1aon* and *3g71* were used for performance tests before [KGE11; LBPH10], while the datasets *3iyj-poly*, *3kz4-poly* and *3iyn-poly* are some of the largest ones available today. Note that this work is the first to attempt computation and visualization of the SES for molecules of such sizes. Performance tests were done on a single machine containing a Intel i9-7900x CPU, a Nvidia Titan Xp GPU and 64 GB of main memory. All surface rendering performance tests were conducted with the resolution 1920×1080. MegaMol was used to load the datasets, setup the OSPRay geometry and render the obtained image with OpenGL.

5.1 Contour Buildup

Section 3.3.2 and Section 4.1 describe the contour buildup algorithm and its implementation details. Figure 5.2 shows the SES computed for the molecule *1ogz*. As the algorithm heavily relies on the usage of floating point numbers, the question for single or double precision floats arises. While single precision floats are beneficial for performance, both Lindow [Lin10] and Totrov and Abagyan [TA96] recommend the usage of double precision floats for computation as it reduces the amount of singularities occurring. It was found that

Table 5.1: The molecular datasets used for the performance tests.

Dataset	Description	#Atoms
1vis	Mevalonate kinase	2,482
1aon	Asym. chaperonin complex	58,674
3g71	Bruceantin	90898
3iyj-poly	Bovine papillomavirus capsid	1.35M
3kz4-poly	Rotavirus capsid	3.24M
3iyn-poly	Human Adenovirus capsid	5.97M

[1] https://www.rcsb.org/

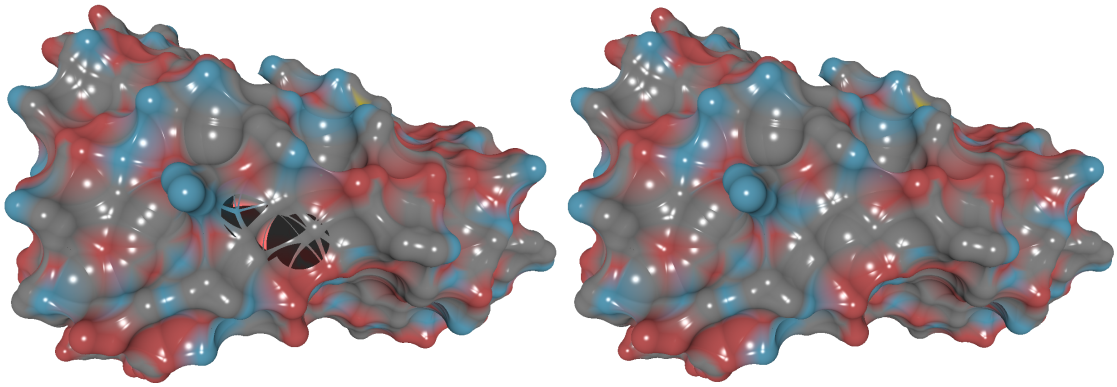

Figure 5.1: SES of the molecule *1vis* with probe radius $R_{\text{probe}} = 3.0\,\text{Å}$. Left: Surface computed with single precision floats. Right: Surface computed with double precision floats.

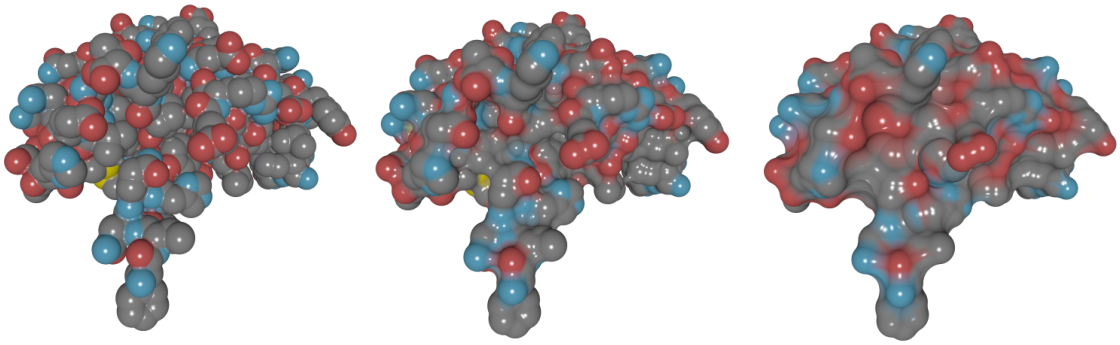

Figure 5.2: SES surface of the molecule *1rwe*, with probe radii $R_{\text{probe}} = 0.25\,\text{Å}$, $R_{\text{probe}} = 1.0\,\text{Å}$ and $R_{\text{probe}} = 2.0\,\text{Å}$, from left to right.

in some cases, single precision floats lead to errors in the surface as can be seen in Figure 5.1, which do not occur with double precision floats. Therefore, the surface computation was done with double precision floats. All contour buildup tests, if not otherwise stated, were conducted with double precision floats.

5.1.1 Vectorized Contour Buildup

As described in Section 4.2, the basic contour buildup algorithm was vectorized. Unfortunately, the tried variants (full and partial vectorization) are slower than the scalar contour buildup implementation. Tests that use vectorized code use a vector width of four, which is most optimal for double precision computations[2]. The AVX2 instruction set was used (ISPC compiler target option *avx2-i64x4*). Table 5.2 compares the scalar circle

[2]See ISPC documentation at https://ispc.github.io/ispc.html.

Table 5.2: Single thread performance of scalar versus partially vectorized, versus fully vectorized circle computation in seconds. The probe radius was chosen as $R_{\text{probe}} = 1.4\,\text{Å}$.

Dataset	Scalar	Part. Vec.	Vectorized
1vis	0.0242	0.0288	0.0284
1aon	0.5456	0.6470	0.6442
3g71	0.9291	1.1180	1.1366
3iyj-poly	12.2856	14.4323	14.2864

Table 5.3: Single thread performance of scalar versus partially vectorized, versus fully vectorized circle computation in seconds. The probe radius was chosen as $R_{\text{probe}} = 3.0\,\text{Å}$.

Dataset	Scalar	Part. Vec.	Vectorized
1vis	0.0919	0.1439	0.1379
1aon	2.0750	3.2855	3.1523
3g71	3.6921	6.1177	6.0568
3iyj-poly	53.5626	84.9795	85.4830

computation with the partially vectorized and the fully vectorized computation for probe radius $R_{\text{probe}} = 1.4\,\text{Å}$. Since a larger probe implies more neighboring atoms, vectorization overhead might be mitigated. However, Table 5.3 shows that a larger probe radius also does not result in the vectorized implementation surpassing the scalar implementation.

In an alternative implementation the arc computation phase of the contour buildup algorithm was partially vectorized. Again the scalar implementation is fastest, as can be seen in Table 5.4 and Table 5.5. In all cases, the fully vectorized implementation behaves similarly to the partial vectorized ones, despite allowing concurrent computation of the contour of up to four spheres at once. This indicates that vectorization is not well suited for complex algorithms such as the contour buildup due to diverging execution paths and asymmetric work loads per program instance, which lead to stalling, as described in

Table 5.4: Single thread performance of scalar versus vectorized arc versus fully vectorized computation, in seconds, for probe radius $R_{\text{probe}} = 1.4\,\text{Å}$.

Dataset	Scalar	Part. Vec.	Vectorized
1vis	0.0156	0.0233	0.0212
1aon	0.3836	0.5515	0.4923
3g71	0.6185	0.5515	0.8108
3iyj-poly	8.7169	12.2376	12.0181

Table 5.5: Single thread performance of scalar versus vectorized arc computation, in seconds, for probe radius $R_{\text{probe}} = 3.0\,\text{Å}$.

Dataset	Scalar	Part. Vec.	Vectorized
1vis	0.0219	0.0317	0.0305
1aon	0.5146	0.7449	0.7381
3g71	0.8552	1.2143	1.2928
3iyj-poly	12.4902	18.2271	19.4183

Table 5.6: Performance comparison of contour computation where every atom's data is individually allocated versus the blocked memory implementation, in seconds. The contour was computed with probe radius $R_{\text{probe}} = 1.4\,\text{Å}$.

Dataset	Individual	Blocked
3iyj-poly	9.92	5.10
3kz4-poly	36.07	11.98
3iyn-poly	107.04	21.93

Section 4.2. Comparing the results for probe radius $R_{\text{probe}} = 1.4\,\text{Å}$ and $R_{\text{probe}} = 3.0\,\text{Å}$, the computation effort for the arc computation phase only rises slightly, while the effort for the circle computation phase increases significantly. The low level of parallelism of just four program instances produces more overhead than performance gains, as opposed to highly parallel environments such as GPUs where this becomes less of an issue. In general, most effort is spend in the first phase for large probe radii. Therefore, further improvements for the algorithm should focus on this phase in the future.

5.1.2 Scalar Contour Buildup

Due to the performance issues with the vectorized implementations, further development was focused on the scalar implementation. For large datasets, memory management becomes an issue. The optimization described in Section 4.1.4 leads to vastly improved contour buildup execution times, as seen in Table 5.6. For the largest dataset *3iyn-poly*, computation time is roughly five times faster due to less individual memory allocations and deallocations. The scalar implementation performance of the entire contour buildup, parallelized over ten cores, and the subsequent generation of the SES's render data is shown in Table 5.7. Here, rough linear scaling with the number of spheres can be observed. This is explained by the fact that the contour computation of a single sphere can be seen as constant for a fixed probe radius, as atoms in molecular dataset have a bounded number of neighbors [Lin10]. The same relation holds if the probe radius is increased, as seen in Table 5.8. Linear scaling also occurs for the contour buildup algorithm's memory usage for both tested probe radii, as shown in Table 5.9 and Table 5.10. Naturally, larger probe radius implies more memory usage, as each atom has more neighbors and therefore

Table 5.7: Performance in seconds of the different phases of the scalar contour buildup (CB) implementation, parallelized over 10 cores using blocked memory. Render Data refers to the creation of the SES from the contour. The contour was computed with probe radius $R_{\text{probe}} = 1.4\,\text{Å}$.

Dataset	Grid	Circles	Arcs	CB	Render Data	Combined
1vis	0.0005	0.0037	0.0018	0.0060	0.0045	0.0117
1aon	0.0045	0.0485	0.0288	0.0819	0.1169	0.2135
3g71	0.0082	0.0843	0.0486	0.1410	0.1758	0.3403
3iyj-poly	0.1150	1.0235	0.6514	1.7900	2.8774	5.0812
3kz4-poly	0.2645	2.5669	1.5808	4.4122	6.5128	12.0472
3iyn-poly	0.5040	5.0644	3.0247	8.5931	11.4219	22.2525

Table 5.8: Performance in seconds of the different phases of the scalar contour buildup (CB) implementation, parallelized over 10 cores using blocked memory. Render Data refers to the creation of the SES from the contour. The contour was computed with probe radius $R_{\text{probe}} = 3.0\,\text{Å}$.

Dataset	Grid	Circles	Arcs	CB	Render Data	Combined
1vis	0.0005	0.0120	0.0023	0.0148	0.0031	0.0203
1aon	0.0035	0.2145	0.0426	0.2606	0.0655	0.3484
3g71	0.0061	0.3694	0.0729	0.4484	0.0917	0.5748
3iyj-poly	0.0835	4.8740	1.0145	5.9721	1.0569	7.6435
3kz4-poly	0.1976	11.7264	2.4466	14.3705	2.4598	18.4427
3iyn-poly	0.3660	24.1389	4.6358	29.1408	3.9013	36.1181

more circles. One disadvantage of the blocked memory approach is observable in the arc memory usage. The memory usage for the arcs does not change with increased probe radius, indicating that the blocked memory approach allocates generous amounts of memory per block. To mitigate this, each block's memory may be reallocated to fit the required data after all its spheres were processed. This yields lower overall memory usage at the cost of performance, as seen in Table 5.11. The amount of observed memory consumption justifies the usage of CPU based implementation for the contour buildup algorithm for large datasets, as current of-the-shelf GPUs may not have enough memory available to hold all computed data. However, for small datasets, the CUDA implementation of Krone et al. [KGE11] is advantageous, as shown in Table 5.12. For the largest tested dataset *1aon*, performance is roughly ten times higher. Note that the CUDA implementation does not allow transparent rendering as internal protruding geometry is not clipped. Additionally, larger datasets such as *3g71* are not supported due to memory constraints.

Table 5.9: Memory usage of the contour buildup, in MB, for probe radius $R_{\text{probe}} = 1.4\,\text{Å}$.

Dataset	Circles	Arcs	Combined
1vis	5.75	6.38	12.13
1aon	135.69	150.71	286.40
3g71	210.21	233.48	443.68
3iyj-poly	3,137.74	3,485.09	6,622.82
3kz4-poly	7,507.72	8,338.84	15,846.55
3iyn-poly	13,818.98	15,348.76	29,167.73

Table 5.10: Memory usage of the contour buildup, in MB, for probe radius $R_{\text{probe}} = 3.0\,\text{Å}$.

Dataset	Circles	Arcs	Combined
1vis	8.63	6.38	15.00
1aon	195.768	150.71	346.48
3g71	312.51	233.48	545.98
3iyj-poly	4,527.14	3,485.09	8,012.22
3kz4-poly	10,846.00	8,338.84	19,184.83
3iyn-poly	23,420.21	15,348.76	38,768.96

Table 5.11: Memory Usage in MB with when memory blocks are reallocated after all corresponding data was written, for probe radius $R_{\text{probe}} = 1.4\,\text{Å}$. Contour buildup (CB) time is in seconds.

Dataset	Circles	Arcs	Combined	CB comp. time
3iyj-poly	1,840.72	1,100.40	2,941.12	2.08
3kz4-poly	4,575.62	2,609.97	7,185.58	5.14
3iyn-poly	8,750.42	4,739.77	13,490.19	10.10

Table 5.12: Comparison of parallelized SES computation versus the CUDA implementation of Krone et al. [KGE11]. Time is given in seconds.

Dataset	CUDA	CPU
1vis	0.0085	0.0117
1af6	0.0124	0.0415
1aon	0.0252	0.2135

Table 5.13: Number of convex spherical patches (CSP), spherical triangles (ST) and toroidal patches (TP), their combined sum and cutting planes of the computed SES with probe radius $R_{\text{probe}} = 1.4\,\text{Å}$.

Dataset	#CSP	#ST	#TP	Combined	#Cutting Planes
1vis	1,483	3,232	4,849	9564	46,809
1aon	39,634	90,782	136,363	266,779	1,319,734
3g71	56,190	128,082	192,449	376,721	1,913,227
3iyj-poly	937,128	2,287,048	3,435,840	6,660,016	31,947,208
3kz4-poly	2,089,305	5,153,772	7,743,515	14,986,592	72,428,209
3iyn-poly	3,566,784	8,213,032	12,343,820	24,123,636	122,485,840

Table 5.14: Number of convex spherical patches (CSP), spherical triangles (ST) and toroidal patches (TP), their combined sum and cutting planes of the computed SES with probe radius $R_{\text{probe}} = 3.0\,\text{Å}$.

Dataset	#CSP	#ST	#TP	Combined	#Cutting Planes
1vis	796	1,668	2,504	4,968	33,952
1aon	17,055	35,898	53,872	106,825	758,219
3g71	22,535	47,450	71,265	141,250	1,041,007
3iyj-poly	275,076	572,080	858,180	1,705,336	11,971,048
3kz4-poly	640,751	1,343,738	2,016,356	4,000,845	29,419,752
3iyn-poly	1,009,224	2,116,888	3,177,108	6,303,220	44,799,048

5.1.3 Render Data

The number of primitives the SES is composed of is shown in Table 5.13 for $R_{\text{probe}} = 1.4\,\text{Å}$ and Table 5.14 for $R_{\text{probe}} = 3.0\,\text{Å}$. For larger probe radii, less primitives are generated. This is explained by the fact that more atoms are buried inside the surface in this case, resulting in less, but larger primitives. For $R_{\text{probe}} = 1.4\,\text{Å}$ roughly 34 cutting planes are used for each convex spherical patch, while for $R_{\text{probe}} = 3.0\,\text{Å}$, the number is roughly 44. This corresponds to the average number of neighbors of each atom as the planes are computed from the intersection circles computed by the contour buildup algorithm.

The time required for the generation of the render data is given in Table 5.7 and Table 5.8. This operation is more expensive than the contour buildup, which results from the fact that this computation is sequential except for the singularity computation of the spherical triangles. Table 5.15 shows the memory consumption of all data generated from the contour required for rendering. The required memory for all render data is lower than the corresponding memory usage of the circles and arcs.

Table 5.15: Memory usage for render data for SES, in MB, for probe sizes $R_{\text{probe}} = 1.4\,\text{Å}$ and $R_{\text{probe}} = 3.0\,\text{Å}$.

Dataset	$R_{\text{probe}} = 1.4\,\text{Å}$	$R_{\text{probe}} = 3.0\,\text{Å}$
1vis	3.30	1.87
1aon	91.78	42.21
3g71	130.63	58.16
3iyj-poly	2312.83	691.37
3kz4-poly	5186.58	1669.62
3iyn-poly	3953.65	2549.50

Table 5.16: Rendering Performance in frames per seconds, for surfaces with $R_{\text{probe}} = 1.4\,\text{Å}$. Sphere tracing used threshold $\epsilon = 0.0001$. The corresponding renderings are given in Figure 5.5.

Dataset	#Primitives	Ferrari	Sphere Tracing
1vis	9,564	100.6	74.2
1aon	266,779	71.2	48.8
3g71	376,721	87.6	59.5
3iyj-poly	6,660,016	26.9	23.1
3kz4-poly	14,986,592	31.2	20.6
3iyn-poly	24,123,636	26.5	17.7

5.2 Surface Rendering

The intersection computation for convex spherical patches and spherical triangles is numerically stable. This is not the case for the toroidal patches, which is a quadric surface. All tested methods show visual artifacts in the form of missed intersections as seen in Figure 5.3. These artifacts are especially visible if the surface is viewed from the inside of the SES against a bright background. The severe artifacts of Bairstow's method discourages its usage. Ferrari's approach and sphere tracing show occasional missed intersections. Sphere tracing with even smaller ϵ removes most of these artifacts. The rendering performance of Ferrari's approach and sphere tracing is shown in Table 5.16. Figure 5.5 shows the resulting renderings. If infrequent pixel artifacts are tolerated, the Ferrari approach shows superior performance, otherwise, sphere tracing with small threshold ϵ is required. Ferrari's approach suffers from another kind of artifacts seen in Figure 5.4, which only appear if viewers are very close to toroidal patches. Again the sphere tracing method is useful if these artifacts are not acceptable. The advantage of spatial acceleration structures such as Embree's BVH result in a sublinear drop in performance for the largest datasets, as large numbers of primitives are excluded from intersection computations as they are occluded by geometry closer to the viewer. GPU based ray casting solutions do not have this property, as all possible primitives are always drawn.

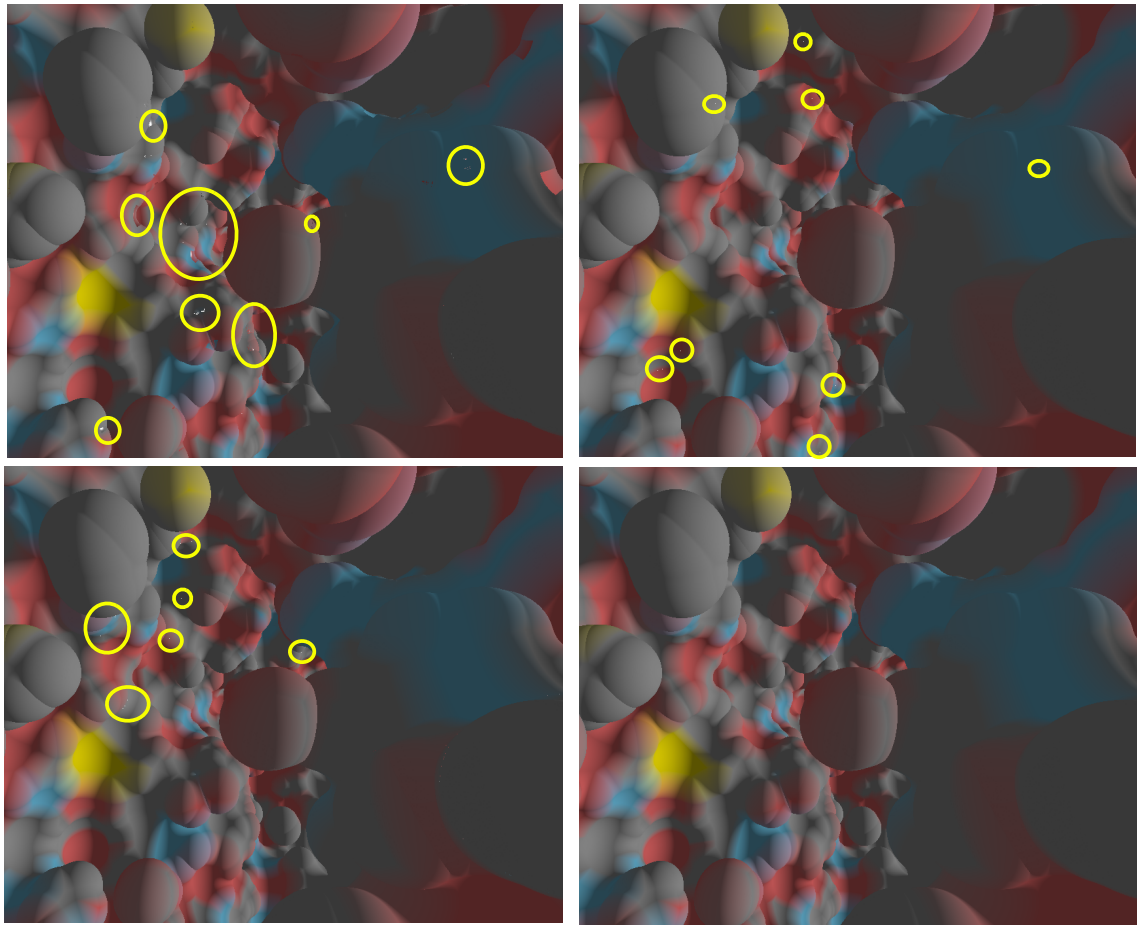

Figure 5.3: View located inside the surface of the molecule *1vis*. From left to right, top to bottom: Bairstow's method, Ferrari's approach, sphere tracing with $\epsilon = 0.001$ and sphere tracing with $\epsilon = 0.0001$. All marked positions show artifacts. Bairstow's method results in most artifacts. Ferrari's approach and sphere tracing with $\epsilon = 0.001$ are acceptable if small pixel errors are negligible. Sphere tracing with even smaller epsilon is able to remove all artifacts in this view.

OSPRay offers the capability to render datasets with shadows and ambient occlusion to improve the viewer's spatial impression of the dataset. Table 5.17 shows performance results if these techniques are applied. Figure 5.6 shows the resulting renderings. Compared to simple local lighting shown in Figure 5.5, each molecule's structure becomes easily recognizable. As one additional ray is cast for each shadow and AO sample a performance decrease by a factor of at least $\frac{2}{3}$ is to be expected. Additionally, the diverging nature of hemisphere sampling to compute the AO weight further reduces performance. Again, larger datasets show the sublinear drop in performance resulting from the BVH.

Transparency rendering naturally reduces performance as rays cannot be terminated at the first intersection they encounter. This reduces the benefit of Embree's BVH. Additional AO computations required in the AOOM technique result in further diminished speed,

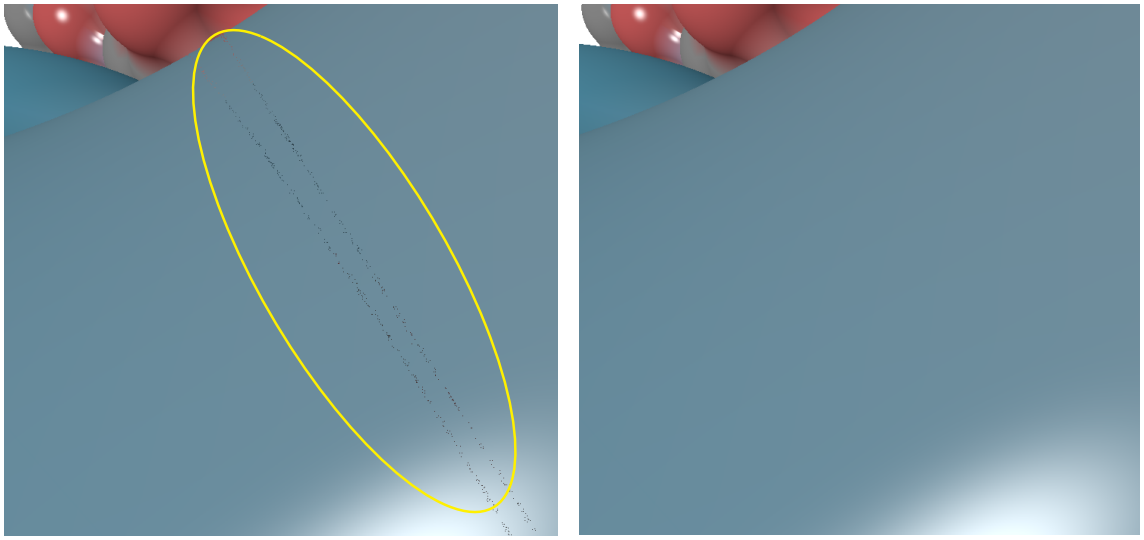

Figure 5.4: If the viewer is very close to a toroidal patch, banded artifacts appear if Ferrari's approach is used (left). These artifacts do not appear with sphere tracing (right).

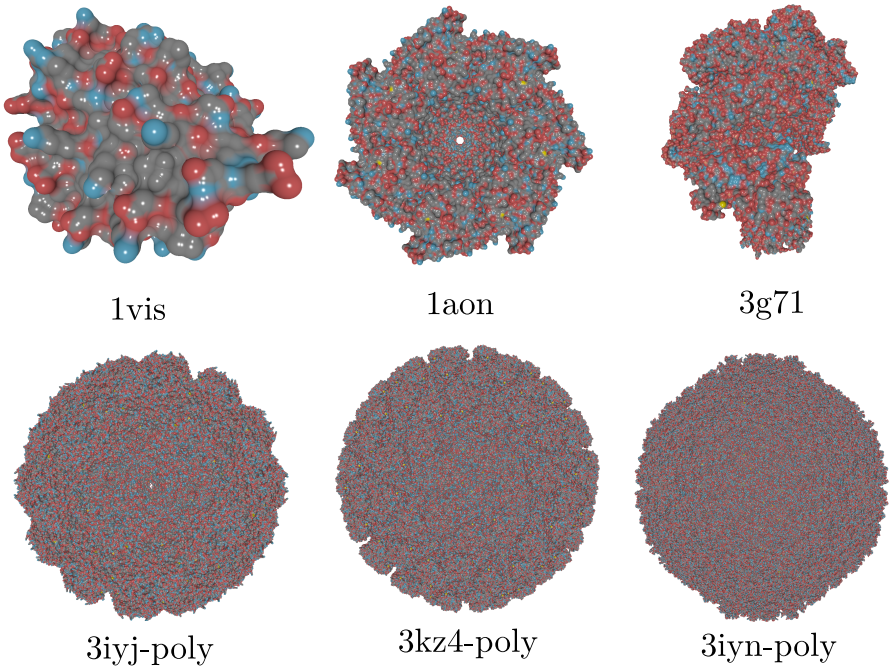

Figure 5.5: Simple local lighting renderings corresponding to the results given in Table 5.16

Table 5.17: Rendering performance in frames per seconds, for surfaces with $R_{\text{probe}} = 1.4\,\text{Å}$. OSPRay's "scivis" renderer was used to compute AO and shadows, each with one sample ray per intersection. The AO sample has unbounded distance. Distant light and ambient light were used for illumination. The corresponding renderings are given in Figure 5.6.

Dataset	**#Primitives**	**Ferrari**
1vis	9,564	25.3
1aon	266,779	13.1
3g71	376,721	16.9
3iyj-poly	6,660,016	6.1
3kz4-poly	14,986,592	5.4
3iyn-poly	24,123,636	4.7

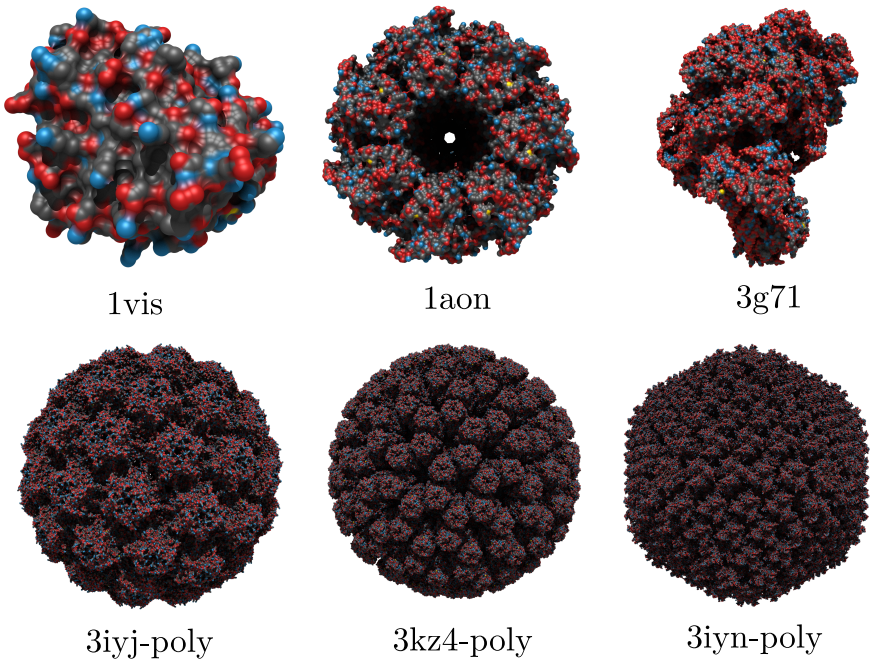

Figure 5.6: Renderings with shadows and ambient occlusion corresponding to the results given in Table 5.16. The molecule structure is easily visible to viewers.

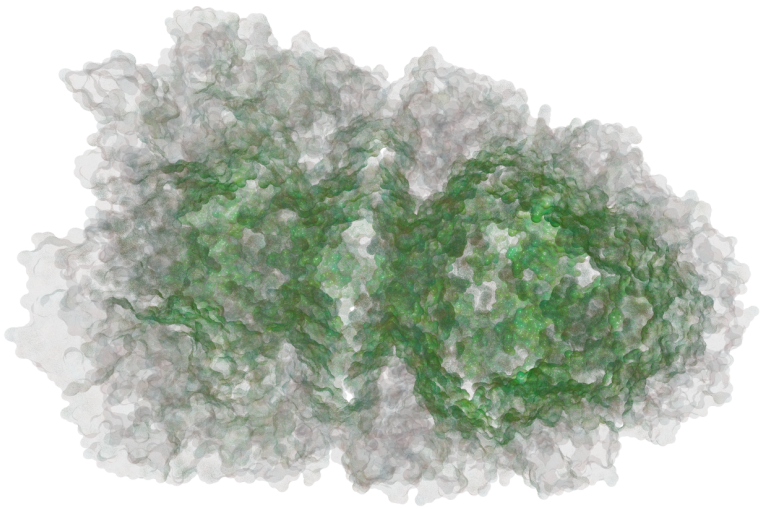

Figure 5.7: For probe radius $R_{\text{probe}} = 1.4\,\text{Å}$, the SES of the molecule *1aon* contains a straight main tunnel through the center with several openings to the side. Equation 4.32 and the AO hemisphere with radius 100 Å were chosen to capture large cavities. To reduce noise introduced by close surface samples, $\rho = 8$ and $\tau = 0.4$. While the main tunnel is visible, its actual shape is difficult to read.

as AO samples have to be computed at each encountered intersection. Table 5.18 shows the performance for both methods. Figure 5.9 shows the renderings obtained from these techniques. The results indicate that these visualization techniques work best for smaller datasets, if interactivity is required. Compared to the original AOOM approach presented by Borland [Bor11], the implemented technique requires no precomputation but does not allow the usage of smoothed ambient occlusion weights. Further, the true analytic SES is used instead of a triangulated representation. While it would be possible to map AO weights from the surface to textures via parametric spherical and toroidal coordinates, distortions introduced by the mappings might become an issue. If large numbers of primitives are present, memory usage for such textures may become a concern as well. The effects of the visibility functions given in Section 4.5 are shown in Figure 5.8. For molecules with high numbers of cavities the visualization becomes cluttered due to many cavities visually overlapping, which makes analysis of individual structures difficult. Additionally, noise resulting from AO sampling further contributes to this issue, as seen in Figure 5.7. Smoothing in surface space could reduce this, however this is difficult to implement in the proposed technique, as it requires knowledge about neighboring surface points.

While OSPRay's pathtracer does not allow rendering of the SES with interactive framerates, publication-ready images can be computed. Figure 5.10 shows renderings computed with path tracing for different materials.

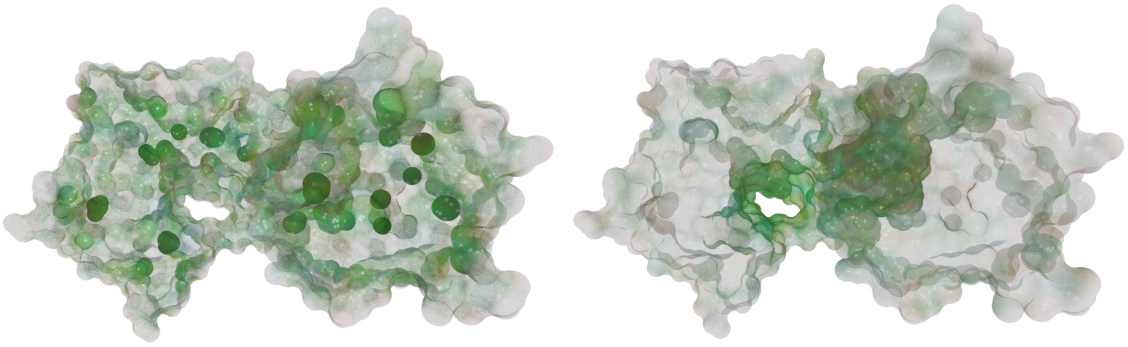

Figure 5.8: For probe radius $R_{\text{probe}} = 1.4\,\text{Å}$, the SES of the molecule *4dfr* contains a tunnel and several small cavities on the inside and outside. Equation 4.31 (left) and Equation 4.32 (right) were used to visualize the cavities. Both images use $\tau = 0.5$. For the left image, the maximum AO sample distance was chosen as 5 Å, while it was chosen as 15 Å in the right image. While $\rho = 1$ gives linear scaling of the visibility in the left image, $\rho = 4$ reduces influence of close AO samples in the right image.

Table 5.18: Rendering Performance in frames per seconds, for surfaces with $R_{\text{probe}} = 1.4\,\text{Å}$ while using transparency rendering and the AOOM technique. Performance is given for the Ferrari algorithm. Transparency rendering uses opacity $\alpha = 0.5$. AOOM rendering uses opacity $\alpha = 0.1$ and color blending weight $\beta = 0.75$. Equation 4.31 was used in the visibility function. The blend color C_{blend} was set to pure green. One AO sample was computed per frame, for a maximum distance of 5 Å. Larger datasets do not give interactive performance. Renderings are given in Figure 5.9.

Dataset	**#Primitives**	**Transparency**	**AOOM**
1vis	9,564	23.4	9.1
1aon	266,779	8.8	2.7
3g71	376,721	10.0	2.7

OSPRay supports instancing of geometries. Complex collections of geometry can be replicated millions of times without much memory overhead for the geometries itself, since the actual geometry exists in memory only once. Figure 5.11 shows the result for a number of tests were the number of instances was increased by an order of magnitude with each test. The instances were positioned in a block with uniform side lengths as shown in Figure 5.12. While there is a exponential drop in performance for the first 10^3 instances, performance decrease starts to flatten out with more instances, which can again be attributed to Embree's BVH, which avoids many intersection computations due to occluded geometry. Larger instance numbers were not possible due to the BVH memory requirements exceeding

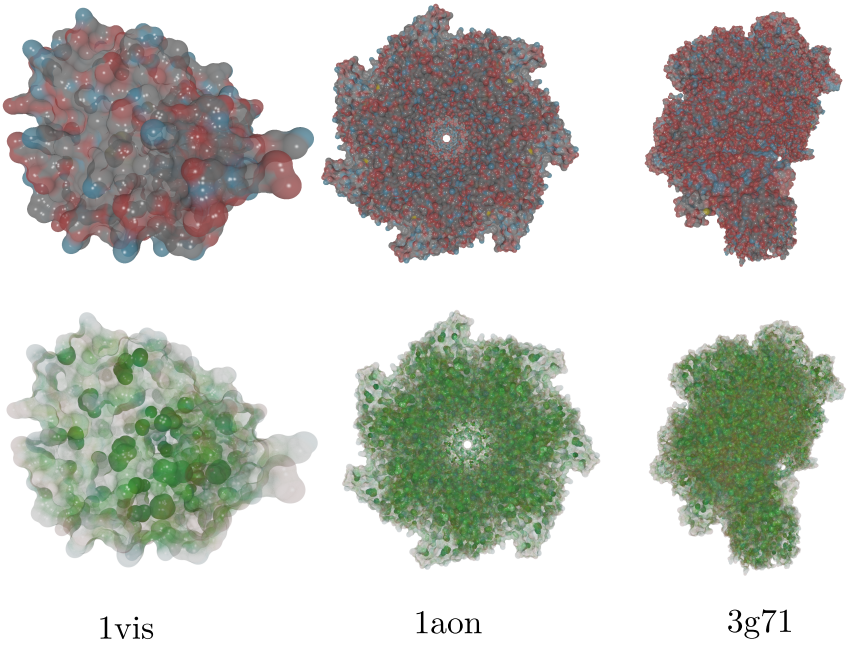

Figure 5.9: Renderings corresponding to the results given in Table 5.18. The top row shows transparency renderings with opacity value $\alpha = 0.5$. Renderings obtained with the AOOM variant are shown in the bottom row. Cavities become easily visible. However, for larger datasets the visualization becomes cluttered.

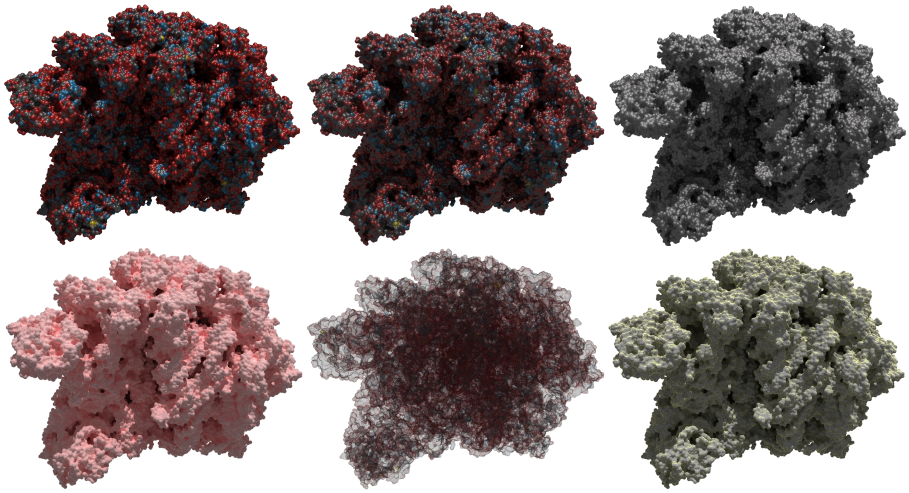

Figure 5.10: Renderings of the molecule *3g71* produced with OSPRay's "pathtracer" renderer, with maximum path length ten. The molecule was illuminated by an ambient light and a directional light source. The images were accumulated for one minute. From left to right, top to bottom: scivis, metal paint, metal, plastic, thin glass and velvet material. In all cases, framerates never rose above 1 FPS.

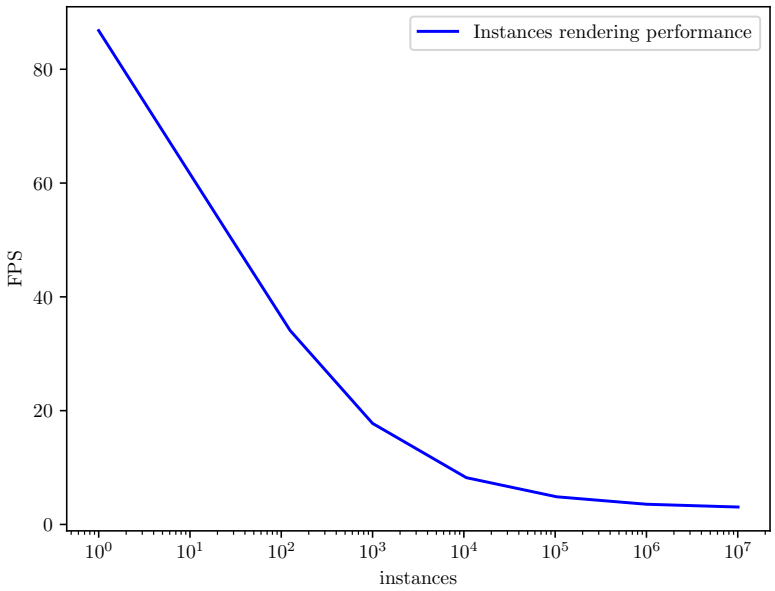

Figure 5.11: Rendering Performance in frames per seconds, for instances of the molecule *1af6* for probe radius $R_{\text{probe}} = 1.4\,\text{Å}$ (38955 SES primitives). Instance numbers were increased roughly by an order of magnitude for each test. Performance is given for the Ferrari algorithm.

available main memory. This suggest the usage of overhead-less acceleration structures such as *P-k-d trees* [WKJ+15]. However, this would requires bounding SES primitives with bounding spheres, as only particle data is supported.

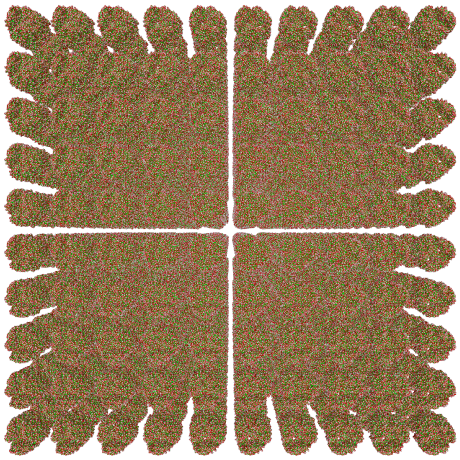

Figure 5.12: The SES of molecule *1af6* instanced $10 \times 10 \times 10$ times in a uniform block.

6 Conclusion and Future Work

The implementation and integration of the SES geometry in the OSPRay framework [WJA+17] presented in this work allows high quality interactive rendering of the surface with CPU ray tracing. Users are able to compute and render the surface such as any other geometry offered by the framework. This makes the implemented geometry available for interactive visualization in HPC clusters lacking GPU capabilities. The integration into OSPRay allows efficient visualization of the surface with global illumination effects and models such as ambient occlusion, shadows or path tracing due to the BVH implementation provided by Embree. Global lighting effects enhance spatial perception of the visualized data and are oftentimes more easily integrated in ray tracing frameworks then GPU based solutions. To obtain the true analytic surface, the contour buildup algorithm was used since it presents ample opportunities for parallelization. While the vectorization attempts for the algorithm did not succeed, a efficient parallelized implementation of the algorithm was achieved nonetheless. This allows the computation and visualization of the SES for molecular dataset composed of millions of atoms, which has not been attempted before. Conducted performance tests show that this requires the presence of large amounts of main memory, currently missing on GPUs. Image quality was achieved by analytic intersection routines for each of the SES's primitive types, while additionally offering the sphere tracing approach for intersection computation for toroidal patches. The usage of sphere tracing is required if occasional pixel artifacts resulting from numerical inaccuracies in Ferrari's approach are not acceptable. Additionally, transparency rendering is supported by removal of all internal protruding geometry. This feature was used to implement an AOOM [Bor11] variant that directly uses the analytic surface without any precomputation to visualize molecule cavities. Visual detection and inspection of such cavities allows domain experts to study interactions between different molecules as these commonly occur in such positions. While the technique works for small molecules with low numbers of cavities, renderings become difficult to read when many cavities overlap.

The current implementation's performance could be further improved by revisiting the vectorization attempts. However, the results obtained in Section 5.1.1 suggest that future work should focus on further restructuring the contour buildup algorithm to reduce branching, or finding another algorithm that is better suited for vectorization. The trilateration approach sketched in Section 4.3 computes intersections of all triplets of neighboring extended spheres from which spherical triangles can be derived directly. Computation of the toroidal patches from this data remains to be solved. While this approach is cubic in nature, careful optimization and implementation on highly parallel hardware such as GPUs may result in faster SES computation than the contour buildup algorithm. The performance results shown in Section 5.1 show that while interactive visualization of the SES is possible for millions of atoms, computation of the surface itself requires multiple seconds on state of the art hardware. Therefore, the contour buildup algorithm (or a

similar algorithm) could be parallelized not only locally on one machine, but additionally distributed over an entire cluster of compute nodes. Lastly, the implementation presented in this work could be used to visualize datasets composed of millions of instanced molecules, similar to the system presented by Le Muzic et al. [LPSV14] interactively by using CPU ray tracing. The instancing test results found in Section 5.2 suggest that this could be possible.

Bibliography

[AGGW15] J. Amstutz, C. Gribble, J. Günther, I. Wald. "An Evaluation of Multi-Hit Ray Traversal in a BVH using Existing First-Hit/Any-Hit Kernels". In: *Journal of Computer Graphics Techniques (JCGT)* 4.4 (Dec. 2015), pp. 72–88. ISSN: 2331-7418. URL: http://jcgt.org/published/0004/04/04/ (cit. on p. 58).

[AM06] S. A. Adcock, J. A. McCammon. "Molecular Dynamics: Survey of Methods for Simulating the Activity of Proteins". In: *Chemical Reviews* 106.5 (May 2006), pp. 1589–1615 (cit. on pp. 15, 37, 38).

[BM08] L. Bavoil, K. Myers. "Order Independent Transparency with Dual Depth Peeling". In: (Jan. 2008) (cit. on p. 57).

[Bor11] D. Borland. "Ambient occlusion opacity mapping for visualization of internal molecular structure". English (US). In: *Journal of WSCG* 19.1-3 (2011), pp. 17–24. ISSN: 1213-6972 (cit. on pp. 42, 43, 80, 81, 96, 101).

[CCW06] T. Can, C.-I. Chen, Y.-F. Wang. "Efficient molecular surface generation using level-set methods". In: *Journal of Molecular Graphics and Modelling* 25.4 (Dec. 2006), pp. 442–454 (cit. on pp. 44, 56).

[CLM08] M. Chavent, B. Levy, B. Maigret. "MetaMol: High-quality visualization of molecular skin surface". In: *Journal of Molecular Graphics and Modelling* 27.2 (Sept. 2008), pp. 209–216 (cit. on p. 40).

[Con83] M. L. Connolly. "Analytical molecular surface calculation". In: *Journal of Applied Crystallography* 16.5 (Oct. 1983), pp. 548–558 (cit. on pp. 15, 40, 42).

[Dun14] A. Dunn. *Transparency (or Translucency) Rendering*. 2014. URL: https://developer.nvidia.com/content/transparency-or-translucency-rendering (cit. on p. 57).

[EW01] C. W. Everitt, L. Williams. "Interactive Order-Independent Transparency". In: 2001 (cit. on pp. 57, 58).

[Fan86] B. T. Fang. "Trilateration and extension to Global Positioning System navigation". In: *Journal of Guidance, Control, and Dynamics* 9.6 (Nov. 1986), pp. 715–717 (cit. on p. 66).

[Fau96] W. M. Faucette. "A Geometric Interpretation of the Solution of the General Quartic Polynomial". In: *The American Mathematical Monthly* 103.1 (1996), pp. 51–57. ISSN: 00029890, 19300972 (cit. on p. 22).

[FS01] D. Frenkel, B. Smit. *Understanding Molecular Simulation*. 2nd. Orlando, FL, USA: Academic Press, Inc., 2001. ISBN: 0122673514 (cit. on p. 38).

[GIK+07] C. P. Gribble, T. Ize, A. Kensler, I. Wald, S. G. Parker. "A Coherent Grid Traversal Approach to Visualizing Particle-Based Simulation Data". In: *IEEE Transactions on Visualization and Computer Graphics* 13.4 (July 2007), pp. 758–768 (cit. on p. 32).

[GKM+15] S. Grottel, M. Krone, C. Müller, G. Reina, T. Ertl. "MegaMol A Prototyping Framework for Particle-based Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 21.2 (2015) (cit. on pp. 16, 35, 39, 76).

[GKSE12] S. Grottel, M. Krone, K. Scharnowski, T. Ertl. "Object-Space Ambient Occlusion for Molecular Dynamics". In: *Proceedings of IEEE Pacific Visualization Symposium 2012*. 2012 (cit. on p. 38).

[Gla89] A. S. Glassner. *An Introduction to Ray Tracing*. London, UK, UK: Academic Press Ltd., 1989. ISBN: 0-12-286160-4 (cit. on p. 18).

[GP06] C. P. Gribble, S. G. Parker. "Enhancing Interactive Particle Visualization with Advanced Shading Models". In: *Proceedings of the 3rd Symposium on Applied Perception in Graphics and Visualization*. APGV '06. Boston, Massachusetts, USA: ACM, 2006, pp. 111–118. ISBN: 1-59593-429-4 (cit. on pp. 35, 39).

[Gum03] S. Gumhold. *Splatting Illuminated Ellipsoids with Depth Correction.* Jan. 2003 (cit. on p. 38).

[Har96] J. C. Hart. "Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces". In: *The Visual Computer* 12.10 (Dec. 1996), pp. 527–545 (cit. on pp. 18, 21, 57, 71).

[Her95] D. Herbison-Evans. "Solving Quartics and Cubics for Graphics". In: *Graphics Gems V*. Elsevier, 1995, pp. 3–15 (cit. on pp. 22, 24, 26–28, 71).

[HKG+17] P. Hermosilla, M. Krone, V. Guallar, P.-P. Vázquez, À. Vinacua, T. Ropinski. "Interactive GPU-based generation of solvent-excluded surfaces". In: *The Visual Computer* 33.6 (June 2017), pp. 869–881. ISSN: 1432-2315 (cit. on pp. 44, 57, 59).

[Jen06] F. Jensen. *Introduction to Computational Chemistry*. John Wiley & Sons, 2006. ISBN: 0470011874 (cit. on p. 37).

[JPSK16] A. Jurcik, J. Parulek, J. Sochor, B. Kozlikova. "Accelerated visualization of transparent molecular surfaces in molecular dynamics". In: *2016 IEEE Pacific Visualization Symposium (PacificVis)*. IEEE, Apr. 2016 (cit. on pp. 41, 57, 58, 70, 72, 73).

[Kaj86] J. T. Kajiya. "The Rendering Equation". In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: ACM, 1986, pp. 143–150. ISBN: 0-89791-196-2 (cit. on p. 20).

[KBE08] M. Krone, K. Bidmon, T. Ertl. "GPU-based Visualisation of Protein Secondary Structure". In: *Proceedings of TP.CG'08*. 2008 (cit. on p. 38).

[KBE09] M. Krone, K. Bidmon, T. Ertl. "Interactive Visualization of Molecular Surface Dynamics". In: *IEEE Transactions on Visualization and Computer Graphics* 15.6 (Nov. 2009), pp. 1391–1398 (cit. on pp. 15, 18, 32, 42–44, 55, 57, 60, 69, 71, 72, 76).

[KFR+11] M. Krone, M. Falk, S. Rehm, J. Pleiss, T. Ertl. “Interactive Exploration of Protein Cavities”. In: *Computer Graphics Forum* 30.3 (June 2011), pp. 673–682 (cit. on p. 57).

[KGE11] M. Krone, S. Grottel, T. Ertl. “Parallel Contour-Buildup algorithm for the molecular surface”. In: *2011 IEEE Symposium on Biological Data Visualization (BioVis)*. Oct. 2011, pp. 17–22 (cit. on pp. 44, 47, 59, 64, 85, 89, 90).

[KKL+15] B. Kozlikova, M. Krone, N. Lindow, M. Falk, M. Baaden, D. Baum, I. Viola, J. Parulek, H.-C. Hege. “Visualization of Biomolecular Structures: State of the Art”. In: *Eurographics Conference on Visualization (EuroVis) - STARs*. Ed. by R. Borgo, F. Ganovelli, I. Viola. The Eurographics Association, 2015 (cit. on pp. 15, 37–40, 42).

[KKL+16] M. Krone, B. Kozlková, N. Lindow, M. Baaden, D. Baum, J. Parulek, H.-C. Hege, I. Viola. “Visual Analysis of Biomolecular Cavities: State of the Art”. In: *Computer Graphics Forum* 35.3 (June 2016), pp. 527–551 (cit. on pp. 15, 37, 40, 41).

[KKP+13] D. Kauker, M. Krone, A. Panagiotidis, G. Reina, T. Ertl. *Rendering Molecular Surfaces using Order-Independent Transparency*. 2013 (cit. on pp. 16, 57, 58, 72, 73).

[KRS+13] M. Krone, G. Reina, C. Schulz, T. Kulschewski, J. Pleiss, T. Ertl. “Interactive Extraction and Tracking of Biomolecular Surfaces Features”. In: *Computer Graphics Forum* 32.3 (2013) (cit. on p. 42).

[KRZ+17] M. Krone, G. Reina, S. Zahn, T. Tremel, C. Bahnmüller, T. Ertl. “Implicit Sphere Shadow Maps”. In: *IEEE PacificVis - Visualization Notes*. Vol. 4. 2017 (cit. on p. 38).

[LBH14] N. Lindow, D. Baum, H.-C. Hege. “Ligand Excluded Surface: A New Type of Molecular Surface”. In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (Dec. 2014), pp. 2486–2495 (cit. on p. 40).

[LBPH10] N. Lindow, D. Baum, S. Prohaska, H.-C. Hege. “Accelerated Visualization of Dynamic Molecular Surfaces”. In: *Computer Graphics Forum* 29.3 (Aug. 2010), pp. 943–952 (cit. on pp. 15, 32, 40, 44, 55, 57, 59, 60, 64, 71, 72, 85).

[Lin10] N. Lindow. “Dynamische Molekülоberflächen”. MA thesis. Technische Universität Berlin, 2010 (cit. on pp. 43, 45, 47–49, 51, 52, 55, 56, 60, 61, 64, 79, 85, 88).

[LPSV14] M. Le Muzic, J. Parulek, A. Stavrum, I. Viola. “Illustrative Visualization of Molecular Reactions Using Omniscient Intelligence and Passive Agents”. In: *Comput. Graph. Forum* 33.3 (June 2014), pp. 141–150. ISSN: 0167-7055 (cit. on pp. 38, 102).

[MB13] M. McGuire, L. Bavoil. “Weighted Blended Order-Independent Transparency”. In: *Journal of Computer Graphics Techniques (JCGT)* 2.2 (Dec. 2013), pp. 122–141. ISSN: 2331-7418. URL: http://jcgt.org/published/0002/02/09/ (cit. on p. 57).

[MS05] A. Markushevich, R. Silverman. *Theory of Functions of a Complex Variable*. AMS Chelsea Publishing Series Teil 11. AMS Chelsea Pub., 2005. ISBN: 9780821837801 (cit. on p. 27).

[OF03] S. Osher, R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer New York, 2003 (cit. on pp. 17, 18).

[PH10] M. Pharr, G. Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN: 0123750792, 9780123750792 (cit. on pp. 32, 81).

[PM12] M. Pharr, W. R. Mark. "ispc: A SPMD compiler for high-performance CPU programming". In: *2012 Innovative Parallel Computing (InPar)*. IEEE, May 2012 (cit. on p. 33).

[PTRV12] J. Parulek, C. Turkay, N. Reuter, I. Viola. "Implicit surfaces for interactive graph based cavity analysis of molecular simulations". In: *2012 IEEE Symposium on Biological Data Visualization (BioVis)*. Oct. 2012, pp. 115–122 (cit. on pp. 41, 42).

[PTVF92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery. *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing*. New York, NY, USA: Cambridge University Press, 1992. ISBN: 0-521-43108-5 (cit. on p. 29).

[RE05] G. Reina, T. Ertl. "Hardware-accelerated Glyphs for Mono- and Dipoles in Molecular Dynamics Visualization". In: *Proceedings of the Seventh Joint Eurographics / IEEE VGTC Conference on Visualization*. EUROVIS'05. Leeds, United Kingdom: Eurographics Association, 2005, pp. 177–182. ISBN: 3-905673-19-3 (cit. on pp. 38, 39).

[Ric77] F. M. Richards. "Areas, Volumes, Packing, and Protein Structure". In: *Annual Review of Biophysics and Bioengineering* 6.1 (1977). PMID: 326146, pp. 151–176 (cit. on pp. 39, 40).

[RKRE17] T. Rau, M. Krone, G. Reina, T. Ertl. "Challenges and Opportunities using Software-defined Visualization in MegaMol". In: *7th Workshop on Visual Analytics, Information Visualization and Scientific Visualization*. 2017 (cit. on p. 36).

[SM09] P. Shirley, S. Marschner. *Fundamentals of Computer Graphics*. 3rd. Natick, MA, USA: A. K. Peters, Ltd., 2009. ISBN: 1568814690, 9781568814698 (cit. on pp. 18, 20, 31, 32, 58).

[SOS96] M. F. Sanner, A. J. Olson, J.-C. Spehner. "Reduced surface: an efficient way to compute molecular surfaces". In: *Biopolymers* 38.3 (1996), pp. 305–320 (cit. on pp. 44, 45).

[ST09] H. M. Senn, W. Thiel. "QM/MM Methods for Biomolecular Systems". In: *Angewandte Chemie International Edition* 48.7 (Jan. 2009), pp. 1198–1229 (cit. on p. 37).

[SVGR16] R. Skanberg, P.-P. Vazquez, V. Guallar, T. Ropinski. "Real-Time Molecular Visualization Supporting Diffuse Interreflections and Ambient Occlusion". In: *IEEE Transactions on Visualization and Computer Graphics* 22.1 (Jan. 2016), pp. 718–727 (cit. on p. 38).

[SWBG06] C. Sigg, T. Weyrich, M. Botsch, M. Gross. "GPU-based Ray-casting of Quadratic Surfaces". In: *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*. SPBG'06. Boston, Massachusetts: Eurographics Association, 2006, pp. 59–65. ISBN: 3-905673-32-0 (cit. on p. 38).

[TA96] M. Totrov, R. Abagyan. "The Contour-Buildup Algorithm to Calculate the Analytical Molecular Surface". In: *Journal of Structural Biology* 116.1 (Jan. 1996), pp. 138–143 (cit. on pp. 43–45, 47, 48, 52, 56, 85).

[TLP07] R. de Toledo, B. Levy, J.-C. Paul. "Iterative Methods for Visualization of Implicit Surfaces On GPU". In: *Advances in Visual Computing*. Ed. by G. Bebis, R. Boyle, B. Parvin, D. Koracin, N. Paragios, S.-M. Tanveer, T. Ju, Z. Liu, S. Coquillart, C. Cruz-Neira, T. Müller, T. Malzbender. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 598–609. ISBN: 978-3-540-76858-6 (cit. on pp. 21, 72).

[Tur57] H. Turnbull. *Theory of Equations*. University mathematical texts. Oliver and Boyd, 1957 (cit. on pp. 22–24).

[Wei18a] E. W. Weisstein. *Quartic Equation*. 2018. URL: http://mathworld.wolfram.com/QuarticEquation.html (cit. on p. 22).

[Wei18b] E. W. Weisstein. *Vieta's Formulas*. 2018. URL: http://mathworld.wolfram.com/VietasFormulas.html (cit. on p. 25).

[WIK+06] I. Wald, T. Ize, A. Kensler, A. Knoll, S. G. Parker. "Ray Tracing Animated Scenes Using Coherent Grid Traversal". In: *ACM Trans. Graph.* 25.3 (July 2006), pp. 485–493. ISSN: 0730-0301 (cit. on p. 32).

[WJA+17] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Gunther, P. Navratil. "OSPRay - A CPU Ray Tracing Framework for Scientific Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (Jan. 2017), pp. 931–940 (cit. on pp. 15, 32, 35, 67, 101).

[WKJ+15] I. Wald, A. Knoll, G. P. Johnson, W. Usher, V. Pascucci, M. E. Papka. "CPU ray tracing large particle data with balanced P-k-d trees". In: *2015 IEEE Scientific Visualization Conference (SciVis)*. IEEE, Oct. 2015 (cit. on p. 99).

[WMG+07] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, P. Shirley. *State of the Art in Ray Tracing Animated Scenes*. eng. 2007 (cit. on p. 15).

[WWB+14] I. Wald, S. Woop, C. Benthin, G. S. Johnson, M. Ernst. "Embree". In: *ACM Transactions on Graphics* 33.4 (July 2014), pp. 1–8 (cit. on p. 35).

[Yu09] Z. 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*. IEEE, Sept. 2009 (cit. on pp. 44, 57).

All links were last followed on May 10, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature