University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Master Thesis

Computation and Visualization of Cavities in Large Molecule Models

Pilsen 2014 Lukáš Jirkovský

Declaration

I hereby declare that this master thesis is completely my own work and that I used only the cited sources.

Pilsen, May 12, 2014

Lukáš Jirkovský

Abstract

With the advances in biochemistry and drug design in the late years, the analysis of molecular data has seen an increased attention. One of the many properties being researched are inaccessible cavities, which are an empty space inside a molecule, and surface depressions called pockets. To discover such empty space, a spherical probe representing a solvent is often used.

In this thesis we tried to solve the problem of finding and visualizing cavities in large molecules of up to hundreds of thousands of atoms. The algorithm finds approximate cavities in a molecule. To allow quick updates when a probe radius is changed, it uses a preprocessing stage where a Delaunay triangulation of molecule atoms is computed. If a precise shape is desired, an algorithm that uses an additively-weighted Voronoi diagram can be used to improve the shape of cavity.

Because there may be a large number of cavities found for a selected probe radius, the filtering of cavities is also important. To support filtering, new methods to measure cavity properties are introduced in this work.

Contents

1 Introduction

A structure of a molecule may contain various elements that are interesting for example in the drug research. An empty space inside a molecule that is not connected to the surface is one of these elements. We call such empty space a cavity. An element related to cavities is a pocket. Pockets are depressions in the molecule surface. Figure 1.1 shows an example of a cavity and a pocket. A spherical probe representing a solvent is usually used to discover cavities by slipping the probe among the atoms. Changing the probe radius allows us to find cavities of different sizes.

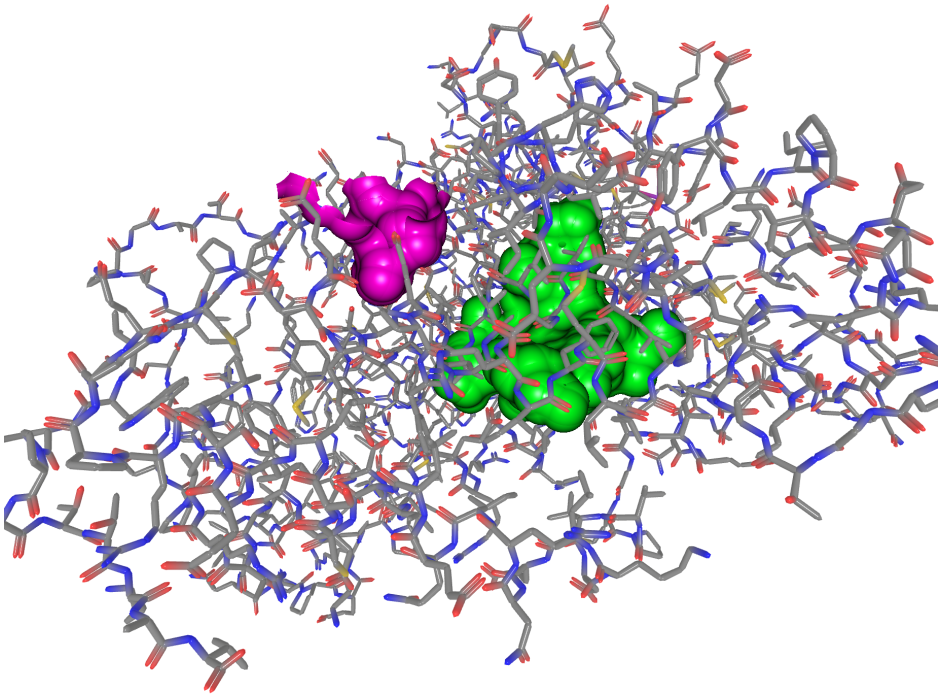


Figure 1.1: A pocket (left) and a cavity (right) in a molecule 1AKD.

Both cavities and pockets take an important role in molecule interactions. Many of these interactions are vital part of biological processes. Therefore their research is important for the understanding of these processes, eventually allowing designing new drugs. A lot of research has been done into locating places where the molecules, such as proteins, interact. These places are called active sites, or binding sites, depending on the molecule. These sites are usually found in large pockets.

As molecules are not stationary, it is possible that an inner cavity becomes accessible from the surface and thus it may become a potential active site. Inner cavities can also affect other molecule properties. For example, they may hold buried residui of other molecules, such as water [34, 37]. These residui can influence the stability of protein structure [37].

The researched molecules, such as these of proteins and ribosomes, may become very large, ranging from dozens to hundreds of thousands of atoms. Processing such large data may be a very time consuming task. Another problem that we face when finding cavities is that many of the cavities are not interesting for research, as they may be too small or complicated. The goal of this work was to find a way to process and visualize such large data effectively, and to design new methods that can be used to filter cavities based on their importance.

In the first chapter, the existing approaches to finding cavities and their visualization will be described. In Chapter 3, the mathematical background necessary for a complete understanding of the presented algorithms will be given. Next, in Chapter 4, an existing solution that we decided to work on will be described. Chapter 5 discusses new methods for finding, visualizing and filtering cavities. The results of our work are given in Chapter 7. Finally, we summarize our findings in Chapter 8.

2 Theory

There are various algorithms that can be used for finding and visualizing cavities in molecules. Finding and visualizing cavities in a molecule is closely related to finding a molecular surface. Many of the algorithms that were designed with finding the molecular surface in mind can be also used for finding cavities either directly, or the surface is used at some point of a specialized cavity-finding algorithm. Apart from that, the molecular surface is usually used for visualizing cavities. Because of this tight connection, both kinds of algorithms will be covered.

Before describing the algorithms themselves, a short summary of the most common types of molecular surfaces will be given, because their knowledge is needed for finding and visualizing cavities. After that, a short summary of the algorithms that are used to find molecular surface will be provided. Finally, some of the algorithms designed specifically for finding cavities will be described.

2.1 Molecular Surfaces

In history, various kinds of molecular surfaces have been introduced. The common ones are a van der Waals surface, a solvent-accessible surface and a solvent-excluded surface. In the following paragraphs, basic information about these surfaces will be provided.

The van der Waals surface [23] is a very simple surface approximation that is defined as a union of spheres. Each sphere represents an atom with a radius defined by a van der Waals radius corresponding to that atom. It is very common model for the visualization of molecules, although it cannot be seen as a real surface. It can be considered as a starting point for the algorithms that are used for finding molecular surfaces.

However, when talking about molecular surfaces, the solvent-accessible surface or the solvent-excluded (Connoly) surface are usually used. For both surfaces, a spherical probe that is swept over the van der Waals surface is introduced. This probe represents a solvent, a common solvent being water with a radius 1.4 Å.

In 1971, Lee and Richards [23] defined the solvent-accessible surface of a molecule. In their definition, the solvent-accessible surface is the surface formed by the van der Waals surface where the radius of each atom was extended by the radius of the probe. Extending the radius is also the algorithm that is commonly used to find the solvent-accessible surface. They also discussed an option of using the algorithm to find cavities in a molecule by eliminating the surface that is connected to the outside of a molecule.

Recently, another kind of surface, the solvent-excluded surface is gaining more attraction. This surface has been popularized by Connolly [10], whose work builds upon the previous work of Richards [30] . It should be noted that some works use the name solvent-accessible surface for this kind of surface, making it difficult to distinguish it from the surface described by [23]. In this work, the term solvent-excluded is strictly used for this type of surface to avoid confusion. This type of surface is created by rolling the probe over the van der Waals surface. The surface consists of three basic shapes – spheres, representing atoms that are also part of the van der Waal surface, spherical triangles, that are formed by a probe touching three atoms, and inner parts of tori, created by a probe revolving around two atoms. In his work, Connolly presented the necessary equations for the analytical description of such surface.

For our work it is important to know some of the basic properties of these shapes. The spheres are clipped by tori in place of their connection. The clipping can be achieved using a clip plane that is defined by the intersection of a torus and a sphere, and that is orthogonal to the axis of the torus. The spherical triangle is a part of sphere that touches exactly three atoms and that is bounded by these atoms. Tori are generated by revolving the probe around two atoms so that the probe always touches each sphere at exactly one point (i.e. the probe has two tangential points in any point in time, one with each sphere). The probe touching the spheres defines an arch between the tangential points. By rolling the probe around the spheres these arches generate a surface. This surface takes shape of a clipped inner part of a torus. In a real molecule the probe usually hits a neighboring sphere during the revolution, causing the surface to be generated only between these tangential points.

Unfortunately, finding the solvent-excluded surface is more complicated than finding other types of surface. Multiple algorithms have been proposed. Some of these algorithms will be covered in the next section.

For better illustration of the differences between the various definitions of surface, see Figure 2.1 and Figure 2.2.

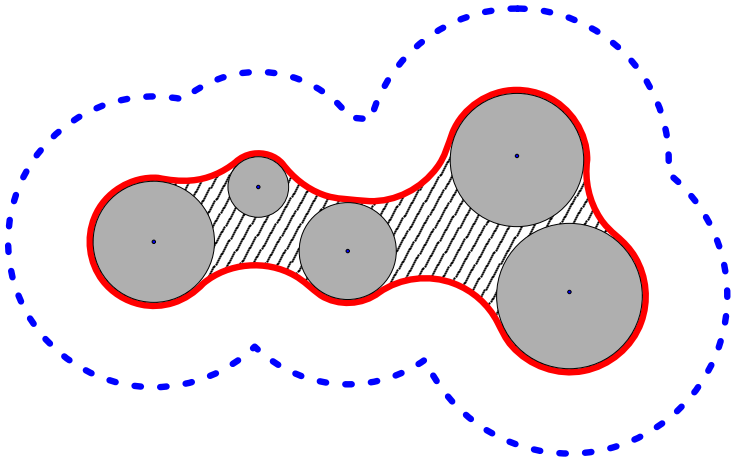


Figure 2.1: Common types of molecular surface in a two-dimensional cross-section. Gray circles represent the van der Waals surface of atoms. The solvent-excluded surface is outlined in red and filled with stripes. The solvent-accessible surface is the blue dashed outline.

2.2 Finding Molecular Surface

The following section describes some of the algorithms that are used to find the molecular surface. These algorithms can find both the outer molecular surface and the surface of cavities. Sadly, they usually does not distinguish between the two, meaning that additional steps must be taken to extract cavities. This usually means determining whether there exist a connection to the outer void. A common problem of these algorithms is that if such an

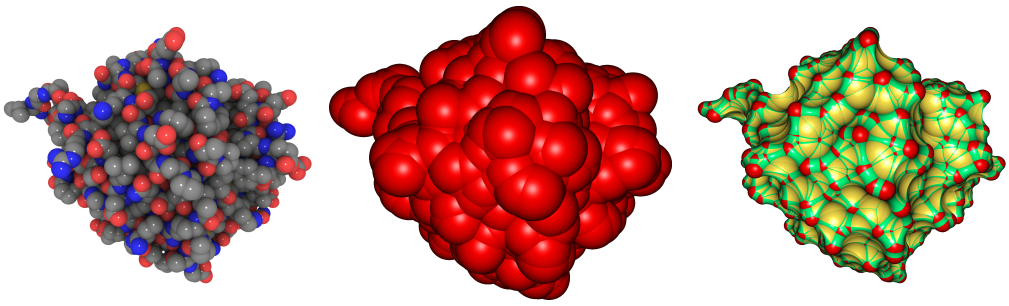


Figure 2.2: Surfaces of a molecule 1CQW. Left: the van der Waals surface. Center: the solvent-accessible surface. Right: the solvent-excluded surface. A probe with radius 3 Å was used for the solvent-excluded and for the solvent-accessible surface.

algorithm is used to find cavities, it provides from little to no information about the cavity itself.

Brute-force Approach

The popular molecular visualization system PyMOL [33] uses a brute-force algorithm that finds a solvent-excluded surface for both cavities and the whole molecule.

The algorithm first finds the van der Waals surface of the molecule. It continues with finding surface points of the solvent atoms that touch the atoms of the molecule's van der Waals surface using a hash map. The surface points are then trimmed. The trimming is done by extending the radius of the molecule's atoms by the radius of the probe and by checking whether the point lies within the space defined by the neighboring enlarged spheres. Finally, the points are used to create a surface mesh.

PyMOL implementation can distinguish the molecular surface and cavities and pockets, but it does not provide much information apart from the visualization, or possibly computing the surface area.

Pseudo-Gaussian Approximation

Pseudo-Gaussian approximation [8] is a method for finding a surface that has been specifically designed to work with large molecular data. Its idea is to replace the solvent-excluded surface with an approximation that uses a scaled Gaussian density function to smooth the transitions among atoms in the van der Waals surface.

The algorithm uses a pseudo-Gaussian density function, which is very similar to the Gaussian density function, but with the added property that it evaluates to 1 when its parameter is equal to the van der Waals radius of an atom, for which the function is evaluated. It also features a parameter called spread, that is used to simulate the effect of changing the probe radius.

The approximation is computed using a molecule placed on a uniform grid. The algorithm itself is as follows: first the spread is computed for a user-given parameter. Then for each atom, the density function is computed

and summed over the uniform grid with small values being ignored, thus reducing the computation only to a limited neighborhood of the atom. After all atoms are processed, the surface, including the surface of cavities, is found by creating a contour of the grid at the level 1. The contour can be found e.g. using the marching cubes algorithm.

It has been proposed to render the surface by superposing the contour on the van der Waals surface, or to use the approximation to construct the surface analytically.

MSMS

When talking about surfaces, we must not forget to mention MSMS [32] because it is a very popular application used to compute solvent-excluded surfaces. MSMS does not implement a single algorithm, but a set of four algorithms. The first algorithm that is utilized by MSMS is an algorithm to find a reduced surface of a molecule. Its output is used by the second algorithm to compute the analytical representation of the solvent-excluded surface. The third algorithm removes singular self-intersections. Finally, the last algorithm builds a triangulated surface.

The first step is computing a reduced surface. The reduced surface is a surface that consists of triangular faces where the probe can touch three atoms, edges where the probe can touch only two atoms without hitting a third atom, and vertices for singular atoms. The principle of the algorithm to find the reduced surface is following: first, a molecule is triangulated using the probe radius for the atom radius. The algorithm starts from an initial triangle at the surface and it recursively tries to roll the probe over the edges of the triangle, discovering the neighboring triangles. The initial triangle can be either hand-picked (useful for examining inner cavities) or a triangle adjacent to the vertex with the lowest x coordinate is used. Also in this step, singular edges where two spherical triangles intersect are identified.

The next algorithm takes the reduced surface as its input and computes the analytical representation of the surface as defined by Connolly [10].

In the third step, the singularities are treated. Three different kinds of singularities are identified. The first involves self-intersecting, opposite-facing spherical triangles generated by the same three atoms. This kind of singularity is responsible for holes in spherical triangles. The second kind of

singularity that is treated here is when a torus is partially clipped by the probe. The last kind of singularities is an intersection of two separate spherical triangles sharing no common atoms. In all cases the identification is based on testing if a probe intersects the triangle. The first two cases are simpler, because we know which probe to use for the test. In the third case, we need to find all probes that can intersect the tested triangle. This is achieved using a tree structure.

The last step that MSMS employs to generate molecular surface is triangulation. The application triangulates the three basic shapes separately, using pre-triangulated templates for spheres and spherical triangles.

2.3 Finding Cavities

This section focuses on the description of some of the prominent approaches to finding cavities in molecules. Many of these algorithms use previously described surface finding algorithms either in one of their steps, or they can use such an algorithm to visualize the found cavities.

SURFNET

SURFNET [22] uses a simple approach based on filling space with spheres. For each two neighboring atoms, it creates a probe that just touches these atoms. If the probe intersects with any other atom, the radius of the probe is reduced until it is no longer intersected by any atom. If the resulting probe is larger than the requested minimal probe, the probe is recorded. This is repeated for all pairs of atoms. All recorded probes are either in cavities or pockets. The results are visualized as a union of spheres representing the recorded probes.

Alpha Shape

Liang et al [26] presented an algorithm based on a three-dimensional alpha shape [12] which is used in the program called CAST. They have also presented means for measuring various properties of cavities [25].

The algorithm starts by finding a weighted alpha shape [11] of molecule's atoms. Then a discrete-flow method is used. In two dimensions, the method finds flows from empty obtuse triangles into acute empty triangles called sinks. After all flows are found, all empty triangles (tetrahedra in three dimensions) are merged to form a cavity.

In [25] they represent a cavity using its corresponding tetrahedra in the alpha shape. In the examples provided in [26], cavities are also visualized by rendering van der Waals surface of atoms forming a cavity.

Grid-based Approaches

There is a multitude of algorithms based on grids. These algorithms usually place the examined molecule on a regular grid, marking each point of the grid with an information whether it is inside or outside of the molecular surface.

Two very similar algorithms that can be used for finding cavities are POCKET [24] and LIGSITE [15]. The POCKET is simpler of these two. It places the molecule on a regular grid. It then moves a probe along each axis of the grid and computes whether an atom center lies inside the probe for each grid point. It also records the nature of the interaction between the probe and atoms. A cavity is identified as a set of points where the probe does not intersect any atom and for which the probe was identified to interact with atoms on each side (i.e. the probe had passed the molecular surface on each side of the point).

LIGSITE uses the scanning approach similar to POCKET. First, it initializes grid points to 0 if they lie outside of the solvent-excluded surface and to -1 if they are inaccessible by solvent. In comparison to POCKET, the probe is moved not only along each axis, but also along diagonals, making it less dependent on the orientation of the molecule. Another improvement is that it increases the grid point value by 1 for each direction for which the point is enclosed by molecule's atoms. The resulting value identifies how buried the point is inside the molecule, where 0 means outside of pocket or cavity and 7 means an inner cavity that is enclosed by atoms in all 7 directions used to scan.

Exner [13] presented an algorithm that uses the solvent-excluded surface and a regular grid to find cavities. The algorithm starts by placing the solvent-excluded surface on a regular grid and marking each grid point as "in" for

points inside the surface and "out" for points outside the surface. Now, the algorithm takes each point marked as "out" as a starting point and checks the neighboring points along each axis up to a distance of 12Å. If there are at least two axes with points marked as "in" in both directions, the point is marked as a cavity point, similarly to the previous algorithms. All the cavity points are then clustered and two operations called contraction and expansion are performed. The contraction removes the cavity points that have non-cavity neighbors from the border. The expansion adds points that have at least one neighbor that is a cavity point. These two operations are performed to remove the small clusters of cavity points that represent small cavities.

Another algorithm called VICE [36] also begins with a grid with points marked 0 for points outside of the molecule and 1 inside. The algorithm takes each point outside of the molecular surface and creates test vectors to all neighboring points up to a defined distance. Again, a cavity is identified by testing whether vectors pass the molecule's atoms. The values in grid are amended by classification of each vector (e.g. the vector passes an atom). The resulting grid values can be used to classify how deep a point is buried as in LIGSITE.

For the visualization purposes, the mentioned algorithms usually use the corresponding part of the surface that was used for the finding of cavities.

Additively-weighted Voronoi Diagram

In our previous work [16] we have presented an algorithm that uses an additively-weighted Voronoi diagram to find cavities. The algorithm will be described in more detail in Chapter 4. It uses preprocessing to construct an additively weighted Voronoi diagram (see Section 3.2). The diagram is used to find the narrowest place among three atoms, whose distance to the atoms is called a bottleneck. The diagram is also used to finds points equidistant to four atoms forming a tetrahedron, which define something like a small cavity.

During the run time the algorithm takes all points for which the distance to the atom surface is large enough for the probe to fit and it tries to slip the probe among atoms using a graph search algorithm using the knowledge about bottlenecks. The advantage of this algorithm is that thanks to the preprocessing it finds cavities for various probes very quickly. The algorithm can be also used to find the outer surface of a molecule and to find pockets.

Output of this algorithm is a set of tetrahedra formed by quadruples of atoms, or the points among them (see the Section 3.2 about their duality). The tetrahedra are very useful for generating the surface. The algorithm for finding the surface finds the boundary faces and uses them to construct an analytic representation of the solvent-excluded surface.

For the visualization a triangulated model of the surface is generated from the analytic surface description.

In this work we have focused on the improvement of this algorithm, because of its speed for variable probe sizes and because the description of cavities it uses is very flexible, making it useful for measuring various cavity properties.

3 Mathematical Background

Before going further with describing the current algorithm and the proposed changes, it is necessary to cover some basic mathematical background. Both the previous solution and the solution proposed in this work are based on an additively weighted Voronoi diagram [28, 18]. In addition, the proposed changes use an ordinary Voronoi diagram [28]. Therefore both diagrams will be described in the following sections, along with their dual counterparts: a quasi-triangulation and a Delaunay triangulation. We will begin with the ordinary Voronoi diagram and the Delaunay triangulation, because these are simpler. The next section will cover the additively-weighted Voronoi diagram and the quasi-triangulation, both of which are extending the ideas introduced by the Voronoi diagram.

3.1 Voronoi Diagram and Delaunay Triangulation

A Voronoi diagram [28] is a tessellation of space (in this work we assume that the space is the Euclidean space) defined for a set containing at least two distinctive points. These points are usually called *generators*, as they "generate" the subdivision of space into several regions. This subdivision of space is created by assigning every location in space to its closest point in the generator set. A region that consists only of points closest to the same generator is called a *Voronoi region*, or a *Voronoi polygon* when the space is two-dimensional or a *Voronoi polyhedron* for an m-dimensional space.

Formally, we can write that for each generator p_i there exists a Voronoi region R_i such that:

$$R_i = \{x : \|p_i - x\| \leq \|p_j - x\|, \forall j \neq i\} \tag{3.1}$$

where $i, j \in \{1, 2, \ldots n\}$ and n is the number of generators.

Boundaries between the Voronoi regions are very important for our work. In the two-dimensional case, a boundary between two regions is called a *Voronoi edge* and it consists of points closest to two generators. We will call these generators *edge generators*. The point where the edges meet is a *Voronoi vertex*. It can be also seen as a boundary point closest to three different

generators. An example of a 2D Voronoi diagram including Voronoi edges and points can be seen in Figure 3.1

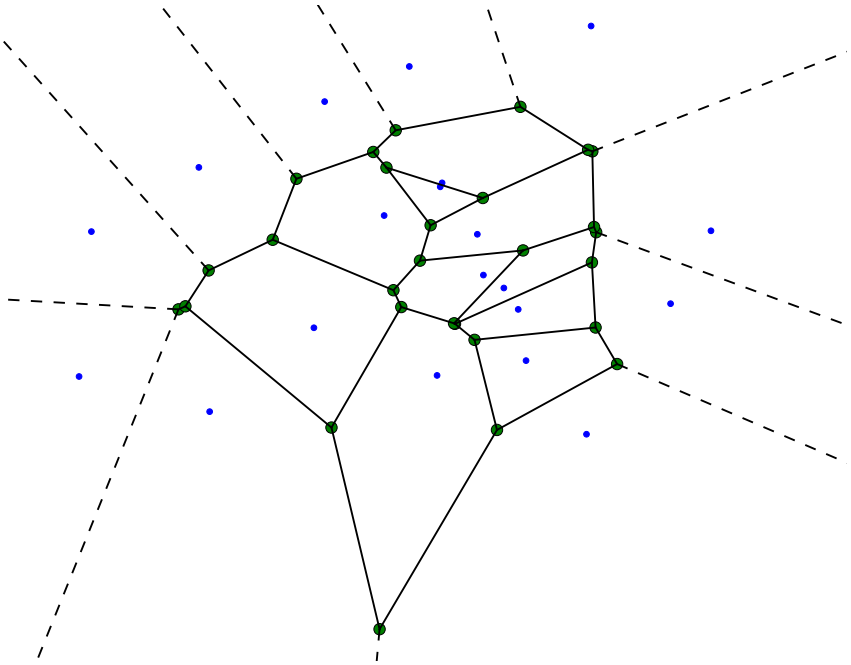


Figure 3.1: A two-dimensional Voronoi diagram of points. The blue dots are generators. The dashed edges extend to infinity.

In the three-dimensional space, a boundary between two regions forms a *Voronoi face*. A Voronoi edge lies on a boundary of three regions and it consists of points equidistant to their generators. Similarly, a Voronoi vertex is a point that is shared among four regions.

The Voronoi diagram is in fact a graph, meaning that various graph algorithms can be used to explore the diagram.

Closely related to the Voronoi diagram is a Delaunay triangulation. The Delaunay triangulation has a few interesting properties, but because these properties are not of great importance for our use, they will be skipped. However, what is important for this work is that it is a dual representation of the Voronoi diagram. That means that the Voronoi diagram can be easily obtained from the Delaunay triangulation and vice versa. It is important especially because the algorithms for constructing Delaunay triangulation are well-established and many implementations exist.

The Delaunay triangulation can be constructed from a Voronoi diagram by connecting each closest pair of generators, i.e. generators sharing an edge in 2D or a face in 3D with an edge. As a result we obtain a triangle for each Voronoi vertex in two dimensions and a tetrahedron in three dimensions. The Voronoi diagram can be constructed from the Delaunay triangulation by applying the procedure in reverse. Figure 3.2 shows this relation between the Voronoi diagram and the Delaunay triangulation.

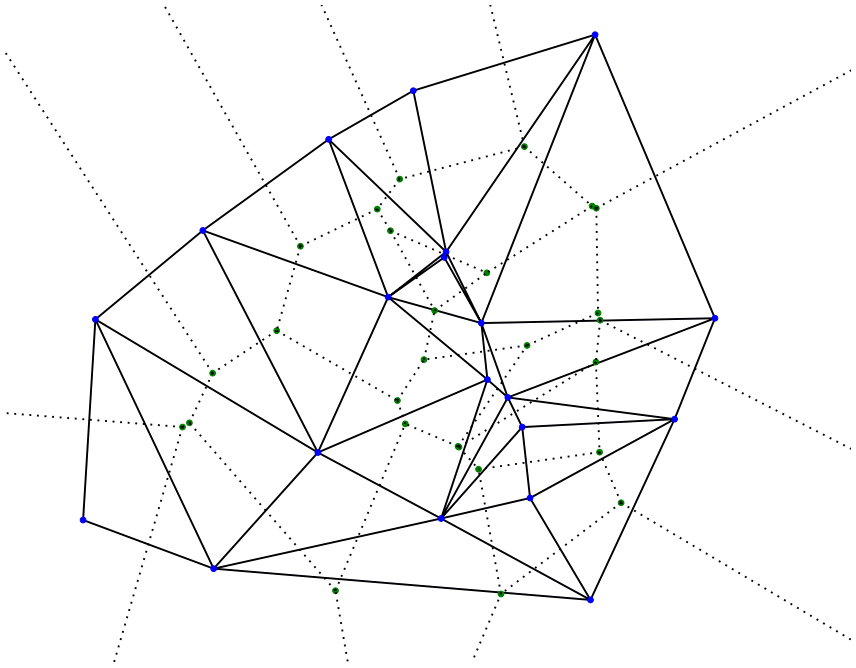


Figure 3.2: A Delaunay triangulation placed on top of a Voronoi diagram.

3.2 Additively-weighted Voronoi Diagram

An additively-weighted Voronoi diagram [28, 18] extends the ideas introduced by the ordinary Voronoi diagram. It is also known as the Apollonius diagram or the Euclidean Voronoi diagram of spheres. The additively-weighted Voronoi diagram is one of many possible generalizations of the ordinary Voronoi diagram.

Instead of using dimensionless point generators that all affect the diagram in the same way, a weight can be added to each point. There are several such

modifications of Voronoi diagram, one of these being the additively-weighted Voronoi diagram. In the additively-weighted Voronoi diagram, the generators are no longer seen as points, but instead as circles in two dimensions and spheres in three dimensions. This is achieved by replacing the Euclidean distance by an *additively-weighted distance* defined as:

$$d_{aw}(\boldsymbol{p}, \boldsymbol{p_i}) = \|\boldsymbol{x} - \boldsymbol{x_i}\| - w_i \tag{3.2}$$

where p is the tested point at position x and p_i is a generator at position x_i with weight w_i. Substituting this distance in equation 3.1, we obtain the following formula for Voronoi region in the additively-weighted Voronoi diagram:

$$R_i = \{\boldsymbol{x} : \|\boldsymbol{x} - \boldsymbol{x_i}\| - w_i \leq \|\boldsymbol{x} - \boldsymbol{x_j}\| - w_j, \forall j \neq i\} \tag{3.3}$$

The region can be imagined as a set of points closest to a surface of the generating spheres. Unfortunately the edges in this diagram are no longer necessarily straight edges, instead they are usually hyperbolic arcs [28]. Figure 3.3 gives an example of the additively-weighted Voronoi diagram in two-dimensions.

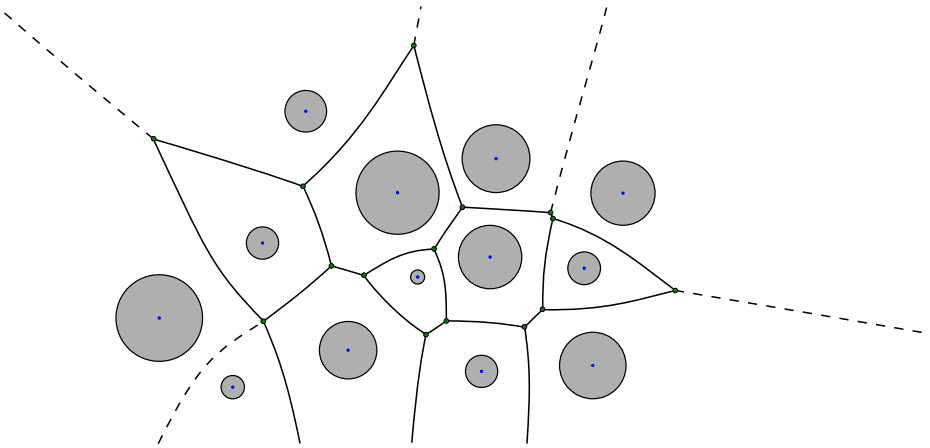


Figure 3.3: An additively-weighted Voronoi diagram in two dimensions. The gray circles are generators.

Also various anomalies can occur in the additively-weighted Voronoi diagram (Figure 3.4). If there is a small sphere (a generator with small weight) among large spheres (generators with large weight), the Voronoi edge can be split into two separate edges. These edges are connected by shorter edges generated by the small generator. Small generator may also generate elliptical edges, which may not be connected to the rest of the diagram and which may even lack a Voronoi point on them. More details about these anomalies can be found in [19]

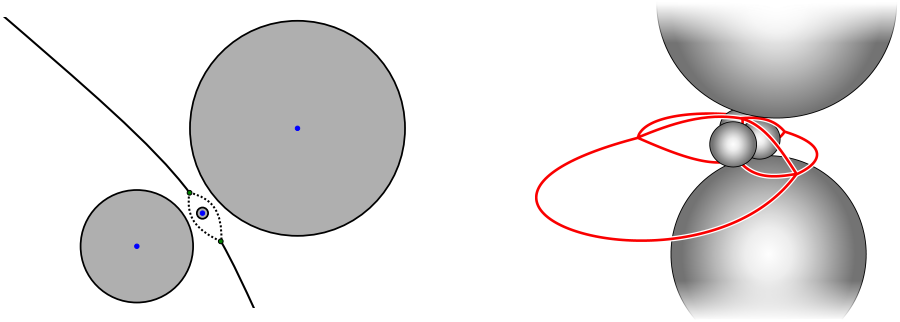


Figure 3.4: Singularities in the additively-weighted Voronoi diagram. Left: a Voronoi edge split by two dotted edges. Right: elliptical edges in a 3D diagram.

Similarly to the ordinary Voronoi diagram, a dual representation exists. It is called *quasi-triangulation* [19]. It can be constructed from the additively-weighted Voronoi diagram in the same way as the Delaunay triangulation was constructed from the ordinary Voronoi diagram. However, compared to the Delaunay triangulation the quasi-triangulation is not necessarily a valid triangulation. In other words, quasi-triangulation is not necessarily formed only by the corresponding simplices such as triangles in 2D and tetrahedra in 3D. This is caused by the anomalies in the additively-weighted Voronoi diagram. For example, an elliptic edge in a three-dimensional diagram diagram is represented by a triangle in the quasi-triangulation. Another example is a split edge, which is, in three dimensions, represented by multiple tetrahedra with two or more triangles in common.

4 Finding Cavities Using the Additively-weighted Voronoi Diagram

In this chapter, the existing approach to finding cavities and their visualization using the additively-weighted Voronoi diagram will be described. The chapter is divided into three sections, describing each part of the process needed to find and visualize cavities.

In the first section, the algorithm to find cavities using the additively-weighted Voronoi diagram will be introduced. The second section describes the finding of cavity surface. Here, the quasi-triangulation will be used in the description, as it is more intuitive than using the additively-weighted Voronoi diagram. However, the real implementation uses quasi-triangulation for both. In the third section, a description of the approach used to visualize cavities will be given.

4.1 Finding Cavities

The method is used primarily to find inner cavities in a molecule. However, it can find cavities that are connected to the outer surface (pockets) and the outer surface, too. The algorithm input is a molecule given as a set of atoms. Each atom is represented as a sphere with an appropriate van der Waals radius. The output is a set of subgraphs of the additively-weighted Voronoi diagram of the molecule. Each subgraph represents one connected cavity in a molecule. This method is implemented in the cavity-finder module that is part of the awVoronoi project [1].

The algorithm has two stages – the first stage is a preprocessing stage that needs to be run only once for a molecule. In the second stage cavities are discovered for a probe of a selected radius. Both stages will be described in more detail in later sections.

The preprocessing leverages the additively-weighted Voronoi diagram. The diagram has two important properties that allow us to find cavities quickly for varying probe size.

First, a Voronoi vertex is equidistant to its four generators. If a spherical

probe with center in the Voronoi vertex is placed so that it touches the generators, the probe does not intersect any other generator, thus defining a minimal spherical cavity with the radius of the probe.

Also, the Voronoi edge in the diagram is equidistant to its generators. This means that the edge is an optimal way among atoms generating the edge. Intuitively, it can be seen that if a probe was slipped away from the edge, the probe would get closer to some atom and possibly collide with it even though there was still a space next to the other atoms. This knowledge is used to find the narrowest point among atoms. The probe can pass among atoms only if it is smaller that the distance from the narrowest point to the generating atoms. We will call this distance a *bottleneck*. We do not need the position of the narrowest point to determine if a probe can pass among atoms, so our algorithm operates only using the bottlenecks.

The second stage, where the cavities are discovered, uses the knowledge about the bottlenecks obtained during the preprocessing. Here we try to slip a probe along the edges of the Voronoi diagram, finding cavities.

Preprocessing

The preprocessing starts by computing the additively-weighted Voronoi diagram of the molecule's atoms. The diagram is used to compute bottlenecks for each edge. Next the Voronoi vertices are sorted according to the distance to their generators. Also, a flag "is outer" is added to each Voronoi vertex. The flag is set to true for all vertices that have at least one edge leading to infinity. It is later used to distinguish the outer void from the inner cavities. In the following paragraphs, we will cover each step in more detail.

To compute the Voronoi diagram, the awVoronoi library [1] is used. The input of this library is a list of weighted points (spheres) that are used as generators. The output is the additively-weighted Voronoi diagram represented in its dual form – the quasi-triangulation.

The bottlenecks are computed for each edge. The algorithm begins by computing the point with minimum distance to the edge generators. However, this point does not necessarily lie on an edge (Figure 4.1). Additional measures must be taken to ensure the bottleneck is computed for a point on the edge. If the point does not lie on the edge, the edge endpoint for which the bottleneck is smaller is used instead. Figure 4.2 show the narrowest points

on each edge.

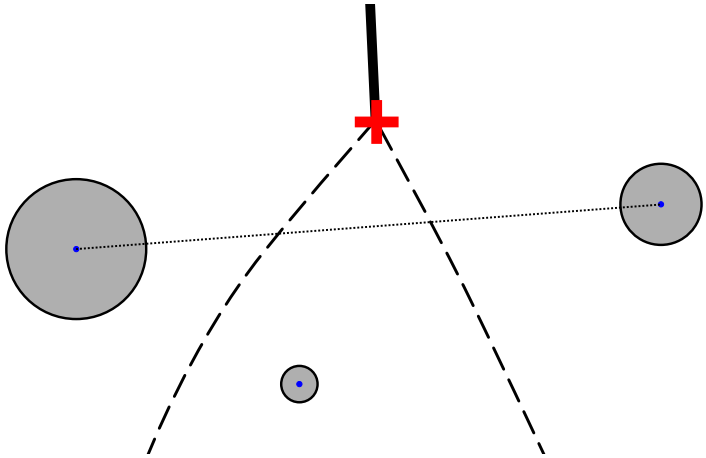


Figure 4.1: The narrowest place between the two large circles lies on the dotted line connecting the circle centers. However, the edge they generate does not intersect this line. Therefore, the closest Voronoi vertex (marked with cross) is used to compute the bottleneck.

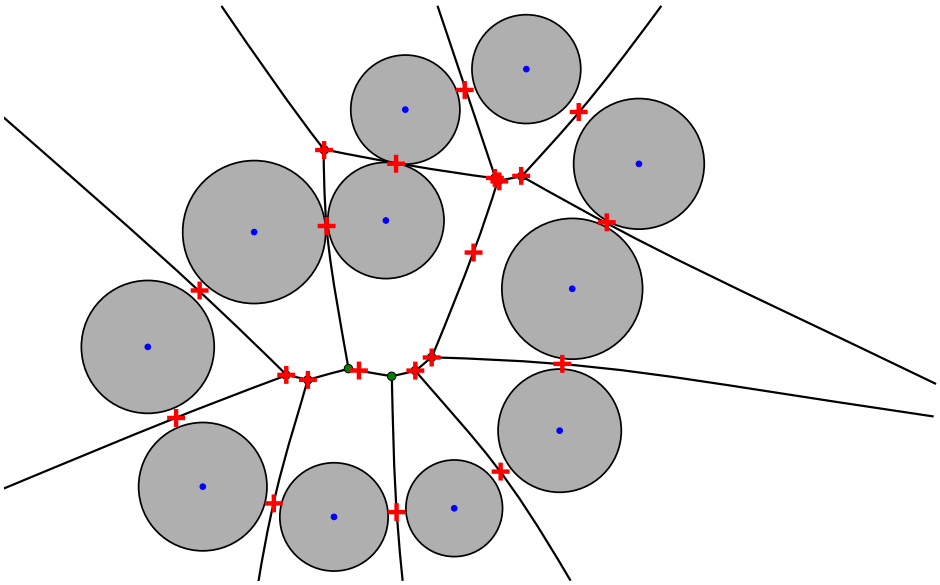


Figure 4.2: The narrowest points on each edge used for bottleneck computation. Each point is marked by a cross.

The check if a point lies on an edge is performed in the following way: first, vectors from the generator with the smallest weight to the edge endpoints are constructed. Second, a test vector from the generator to the tested point is formed. The tested point lies on the edge if and only if the test vector lies withing the angle defined by the vectors to the edge endpoints.

Although the bottlenecks can be already used to find cavities, we improve this approach even further to ensure that it is not necessary to test all vertices when finding cavities to improve the speed. We introduce a *maximal bottleneck* property to each Voronoi vertex, and define it as a distance from the Voronoi vertex to its generating spheres. This distance is always larger than or equal to the edge bottleneck. The maximal bottleneck not only tells us if the probe can fit among atoms, but it also defines a radius of a small cavity formed by the generators, as mentioned earlier. The last step of preprocessing is to sort the vertices using their maximal bottleneck.

Finding Cavities

The second stage is the finding of cavities itself. Its input is a Voronoi diagram, bottlenecks and a sorted list of Voronoi vertices that were obtained in the preprocessing stage. The algorithm is based on the idea of slipping a probe along the edges of Voronoi diagram if there is enough space to do so. As the Voronoi diagram is stored in a graph-like structure of vertices and edges, it can be explored using a graph-traversal algorithm.

The algorithm for finding cavities takes the vertex with the largest maximum bottleneck from the sorted list of vertices. If the maximum bottleneck is larger than the probe radius, a constrained graph traversal algorithm starts (our implementation uses a depth-first search) from this vertex. The graph-traversal algorithm is allowed to move between vertices only if the bottleneck associated with the edge is larger than the probe size. If the bottleneck is too small, the traversal behaves as if there was no edge. This idea is illustrated in two dimensions in Figure 4.3. The Voronoi vertices visited during the traversal form one cavity (Figure 4.4).

The algorithm continues by selecting the next unvisited vertex with the next largest bottleneck from the sorted list of vertices and again starts the graph traversal. This is repeated until the maximum bottleneck of the next vertex in the list is smaller than the probe. Thanks to the sorting performed during preprocessing we are sure that all remaining vertices in the list have smaller maximum bottleneck and therefore they cannot be a part of any cavity.

The result is a list of all cavities in the molecule. However, this includes the outer void, too. To differentiate the outer void from the inner cavities, the algorithm uses the “is outer” flag that was introduced earlier. If a traversal reaches a vertex that has the “is outer” flag set, the flag is set for all vertices

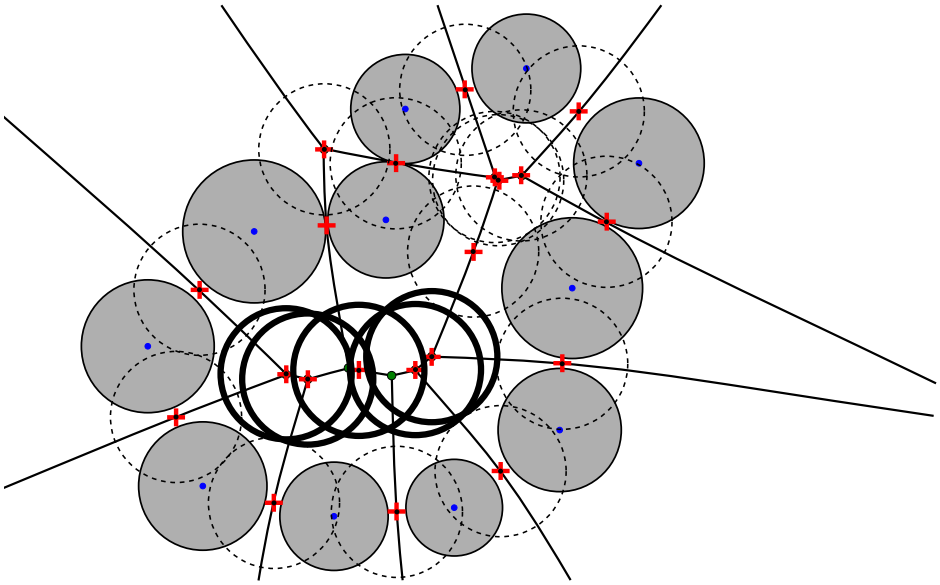


Figure 4.3: Probes in the narrowest points of each edge. The probes that fit have a full outline, while the probes that cannot pass have a dashed outline.

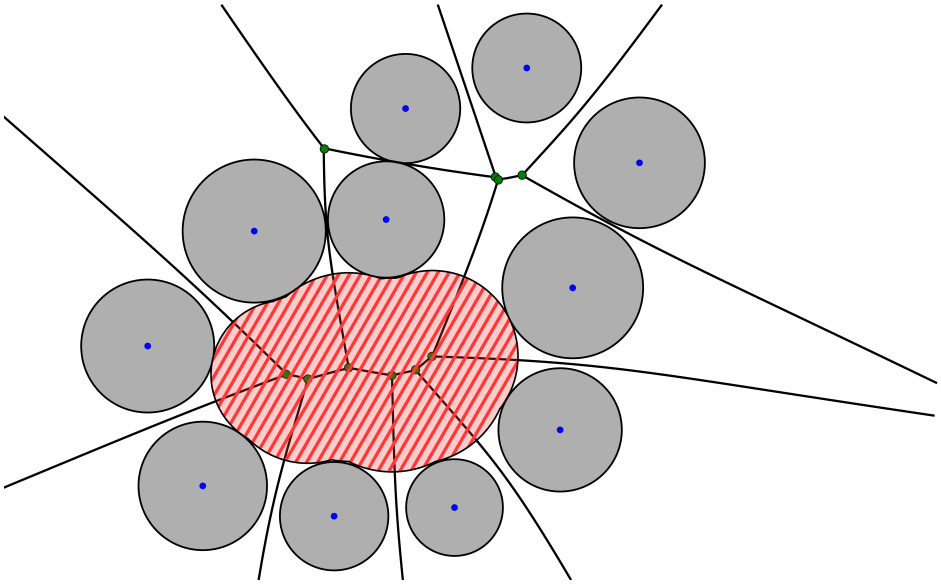


Figure 4.4: A cavity (filled with stripes) approximated as a union of circles.

of the cavity that is being discovered, too. A cavity whose vertices have this flag set is part of the outer void.

To make the algorithm even more flexible, the finding of inner cavities and the finding of outer void can be done in two steps. The first step is executing

the described algorithm using a selected probe radius to find the outer void. In the next step the algorithm is executed again, but this time the vertices are not marked using the "is outer" flag. Instead, the "is outer" flag from the first step is used to restrict the algorithm from entering the outer void. This approach allows finding not only inner cavities, but also pockets and channels that have connection to the surface.

The algorithm currently does not handle the elliptical edges. Fortunately the anomalies found in the additively-weighted Voronoi diagram should not occur in molecular structures, because the differences between radii of various atoms are relatively small.

4.2 Finding Surface

The current solution uses an algorithm designed and implemented by Mgr. Martin Maňák in a plugin for Caver [9]. The input of this algorithm is a quasi-triangulation, a cavity and bottlenecks from the cavity-finding algorithm. The output is a solvent-excluded surface described by a list of spherical triangles, list of tori and a list of spheres. The method is also described in a not-yet published article [27].

For the purpose of the description we will use the quasi-triangulation, as the faces of tetrahedra in the triangulation have direct mapping to the shapes forming the surface and therefore using quasi-triangulation is more intuitive approach. When using a triangulation, a face of cell corresponds to a spherical triangle, edge of a face corresponds to a torus and a vertex corresponds to a sphere. As the quasi-triangulation is dual to the additively-weighted Voronoi diagram, we can easily transform between the two. When the quasi-triangulation is used, a cavity is formed of tetrahedral cells instead of Voronoi vertices. Each face of cell corresponds to a Voronoi edge, meaning that if the a probe can pass along the edge, we can also say that the probe can pass through the face of cell.

The algorithm starts with finding the boundary of a cavity. The boundary consists of triangles on the cavity surface, edges between triangles, and vertices. First, the surface triangles are identified. This is done by testing faces of each cavity cell. If a probe cannot pass through the face, the face is added to the list of surface triangles. A hashing needs to be used to ensure the triangles are not duplicated. Also a list of edges between triangles and a

list of cavity vertices are created in this step. Next, intersections between spherical triangles are solved.

The next step is processing the triangles, edges and vertices to find parameters of spherical triangles, tori and spheres.

A sphere is described using three parameters – its center, the radius, and a list of clip planes. The sphere center is the vertex position. The radius is the van der Waals radius of the atom. The list of clip planes is derived from the edges adjacent to the vertex. Each edge contributes by a clip plane that removes a part of sphere that would by overlapped by the torus generated by that edge.

A torus is described by the two spheres adjacent to the torus, the radius of the probe that was used to find the surface, the initial position of the probe and the angle for which the probe can rotate around the edge before hitting another atom.

Finally, the spherical triangles are described by the center of the sphere on which the triangles lies, the three spheres in the corners of the triangle, the list of clip planes and three vectors from the sphere center to a point of contact between the spherical triangle and a corner sphere. The clip planes are computed from the intersections among spherical triangles (see also the discussion of singularities in MSMS 2.2).

4.3 Visualization

The visualization takes a solvent-excluded surface described by spherical triangles, tori and spheres from the previous step and displays it. The solution uses triangular meshes to model each shape of the surface separately. To reduce the memory requirements, the visualization uses instancing, i.e. there is only one mesh generated for each shape. The mesh is transformed to its final form in the vertex shader.

A sphere uses a unit sphere mesh. In the vertex shader, the sphere is translated to the desired position and scaled according to the van der Waals radius. In the fragment shader, the fragments are clipped using the clip planes.

A spherical triangle uses a mesh in the shape of a subdivided triangle. The vertex shader uses barycentric coordinates to transform the triangle to a

spherical patch of a unit sphere. The patch is then moved to the desired position. The fragment shader clips the unwanted parts of the spherical triangle.

A torus is rendered using a subdivided rectangular grid. The vertex shader transforms this grid to a desired part of a torus. In the fragment shader, only the self-intersecting part is clipped.

One of the problems of a solution using separate meshes for each surface shape is continuity of the surface. It is very difficult to make sure the shape fits together perfectly without artifacts caused by discontinuities.

5 Proposed Solution

In this work I tackled the first and the last step of finding cavities, i.e. the finding cavities itself and the visualization.

While the finding of cavities is already very fast, the construction of the additively-weighted Voronoi diagram in the preprocessing stage is slow. For that reason we focused on finding a way to speed up the preprocessing. The first section presents a possible solution which uses the Delaunay triangulation to find an approximation of cavities, because it is faster to compute than the additively-weighted Voronoi diagram. This approximation can be used as a foundation for computing precise shape using the additively-weighted Voronoi diagram, while effectively reducing the the number of generators used to compute the diagram.

Our second goal was to improve the visualization to handle large molecular structures, ideally to allow real-time interaction. This is described in the second section of this chapter. I have reused the previous work of Mgr. Martin Maňák for finding the cavity surface, as that already works well.

5.1 Finding Approximate Cavities

As the computation of the additively-weighted Voronoi diagram is very time-consuming, we were trying to find a solution that would reduce its cost to minimum. The proposed solution takes advantage of the fact that the size differences among atoms are relatively small. Thanks to this fact, the difference between the additively-weighted Voronoi diagram and the ordinary Voronoi diagram is small (Figure 5.1), making it possible to use the Delaunay triangulation to find approximate cavities. The algorithm finds a set of cavities that is a superset of cavities found when using the additively-weighted Voronoi diagram. If required, the user can request a precise computation of cavities that looks promising. This "shape-improvement" is done on the atoms forming the approximate cavity only, thus reducing the time required to compute the costly additively-weighted Voronoi diagram.

The input and the output of this algorithm are the same as in the previous solution, that is – the input is a set of atoms represented as spheres and the

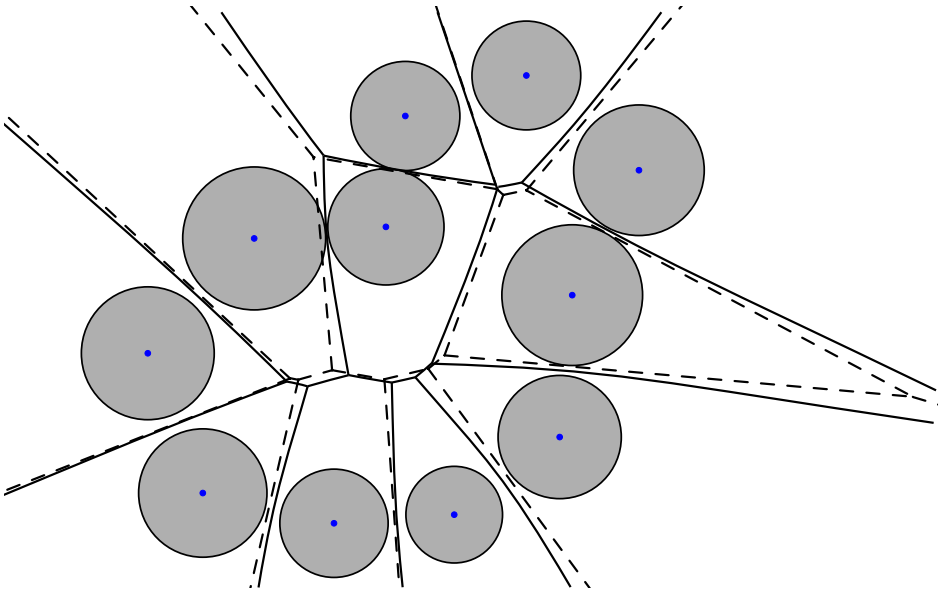


Figure 5.1: Comparison of Voronoi diagrams when the generators have approximately the same size. The additively-weighted Voronoi diagram has a full outline. The ordinary Voronoi diagram using only the circle centers as generators has a dashed outline.

output is a set of cavities. The algorithm operates within three steps. The first step is preprocessing, the second step is finding approximate cavities and the last step is an optional improvement of the shape of the discovered cavities.

In the preprocessing step, a Delaunay triangulation of atoms is computed. For the purpose of computation the atoms are seen as points. Also in this step, bottlenecks are computed. The principle is the same as in the previous solution (see 4.1). However, there is one important difference. While in the previous solution the bottlenecks were computed using the real van der Waals radii of each atom, as the the additively-weighted Voronoi diagram allows that, a single radius must be used for all atoms when using the Delaunay triangulation.

There are many possible options for choosing the radius, but only some of them make sense for the cavity computation. Namely, using a mean of radii of all atoms would give us, in theory, the closest approximation to the precise cavities that can be computed using the additively-weighted Voronoi diagram. However, this means that some of the atoms would be enlarged, causing some cavities to be either omitted completely or to be reduced. A solution is to use a radius of the smallest atom in the molecule. While this approach may be "too optimistic" in that it can find more and larger cavities,

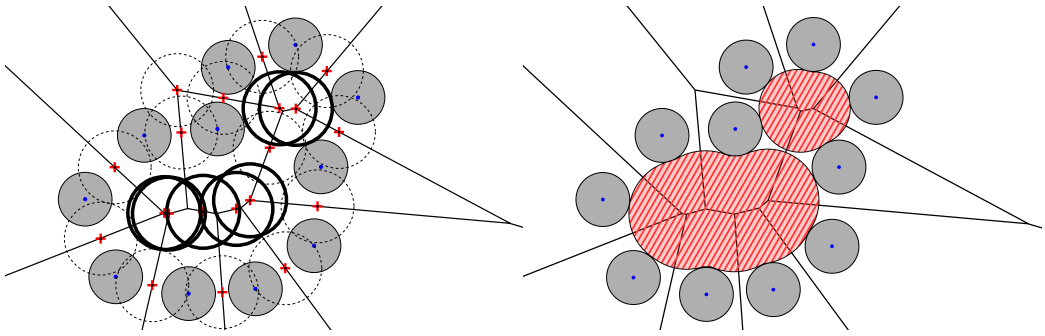


Figure 5.2: Probes (left) and the resulting approximate cavities (right) found using the proposed algorithm. The same generators as in e.g. Figure 4.4 were used with their radius changed according to the algorithm.

it ensures that no cavities are left out or reduced. The mentioned problems can be remedied in the third, optional, step of the algorithm.

The second step is finding cavities itself. The algorithm here is exactly the same is in the previous solution, with one exception – the bottlenecks are the ones computed using the Delaunay triangulation. Figure 5.2 shows probes and computed cavities when the ordinary Voronoi diagram (or the Delaunay triangulation) is used to find approximate cavities. Note that there are two cavities now compared to the Figure 4.4 that used the same generators.

The third step is optional. The input of this step is a cavity that the user wants to improve. The output is one or more cavities computed with the improved precision that replace the input cavity. In this step, the atoms that form a cavity are put into a list. This is done by iterating over all Voronoi vertices and putting their generators into a list of atoms, while employing hashing to avoid duplicates caused by the generators being shared among multiple Voronoi vertices. Once a list of atoms that form the cavity is constructed, the algorithm using the additively-weighted Voronoi diagram, as described in Chapter 4, is executed. Because the number of atoms has been reduced significantly (only the cavity atoms are used instead of all atoms), the time required to compute the additively-weighted Voronoi diagram is much shorter.

As the cavity that is input to the third step is computed using the smallest possible atom radius for all atoms, it is likely that when using a real van der Waals radii some of the generators becomes larger, causing smaller bottlenecks. The result is that the cavity is likely to get reduced in some way. In extreme case, the cavity may be removed completely, however, this happens only with small cavities that bear no real significance. More often, the

boundary of the cavity is reduced, or a cavity is split up into several smaller cavities.

5.2 Visualization

Because the solution presented in Section 4.3 suffered from the number of triangles being rendered and the possible non-continuity among surface shapes, we have decided to implement visualization using a ray casting. This should fix both problems, because the ray casting needs to render only simple bounding objects and it provides a precise analytical description of a surface. Rendering of molecules and molecular surfaces using ray casting has been previously described by e.g. Krone [21].

The input of the visualization are spherical triangles, tori and spheres from the method described in Section 4.2.

Ray Casting

Ray casting is a method that is used to solve various problems by solving ray-surface intersections. It is commonly used for rendering surfaces and solids [31]. For the rendering purposes, ray casting is used to visualize a surface using its analytical description while using a simple bounding object for rendering. Ray casting can be fully implemented on GPU. In our work, we used OpenGL shaders to implement ray casting, so the terminology used in the following text is based on the OpenGL terminology.

The methods described in this section are used in a paper [27] that is being prepared for publication.

The first step is to render a bounding object. For each point on the object, a ray from the camera to the fragment, or image point, is constructed. With the ray casting, this ray is used to compute the intersection between the ray and the desired object in the fragment shader. The computed intersection is then used instead of a real fragment position for computing the lighting. Additionally, it is required to compute a new fragment depth for the z-buffer to work correctly. In OpenGL, the fragment depth is computed using the

formula defined in [17]:

$$gl\_FragDepth = \frac{z(far - near)}{2w} + \frac{near + far}{2};$$

where far and $near$ are positions of clip planes, z is the z coordinate and w is the homogenous coordinate.

Ray Casting of Spheres

To render a sphere using ray casting, a cube is used as a bounding box of the sphere as shown in Figure 5.3. To simplify the computation, the cube coordinates range from $[-1, -1, -1]$ to $[1, 1, 1]$.

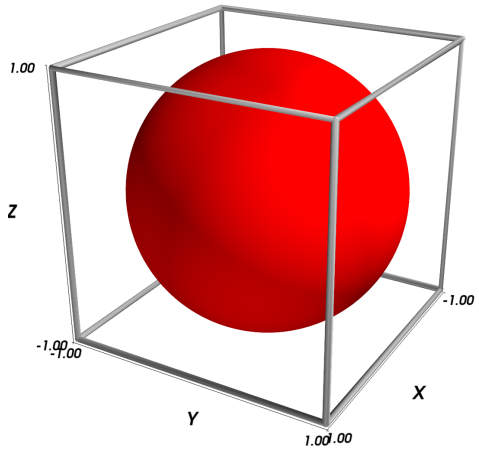


Figure 5.3: A unit sphere enclosed in its bounding box used for ray casting.

In the vertex shader, each vertex position is scaled by the radius of sphere to make sure that the cube is large enough. All computations are carried out in the world space, so the position of the sphere center has to be added to each vertex to obtain the coordinate in the world space. This can be done because the modelview matrix is used only for the view transform. Next, the position of eye in the world space is determined. The position of eye in the view space is $[0, 0, 0, 1]$. To obtain the position in the world space, it needs to be transformed using the inverse of the modelview matrix. After carrying out the necessary multiplication, we will find out that the transformed position is equal to the last column of the matrix. Now, a vector from the eye in

the world space to the sphere center and a vector from the eye to the vertex position in the world space are created. Finally, the vertex position in the world space is transformed using the modelviewprojection matrix.

In the fragment shader, we need to solve the intersection of a ray with a sphere. To solve this, a following formula for the sphere is used:

$$\|\boldsymbol{x} - \boldsymbol{c}\|^2 = r^2 \tag{5.1}$$

and a parametric equation of a line:

$$\boldsymbol{x} = \boldsymbol{o} + t\boldsymbol{d} \tag{5.2}$$

where $\boldsymbol{x}$ is the point of intersection, $\boldsymbol{c}$ is the sphere center, r is the sphere radius, $\boldsymbol{o}$ is the ray origin (the position of eye) and d is the ray direction.

Because the intersection point lies both on the line and the sphere, it is possible to substitute for $\boldsymbol{x}$ and find t:

$$\|\boldsymbol{o} + t\boldsymbol{d} - \boldsymbol{c}\|^2 = r^2 \tag{5.3}$$

substitute $\boldsymbol{l} = \boldsymbol{o} - \boldsymbol{c}$:

$$\|t\boldsymbol{d} + \boldsymbol{l}\|^2 = r^2 \tag{5.4}$$

expand the left side to get rid of the norm and subtract the right side to obtain a quadratic equation:

$$t^2\boldsymbol{d} \cdot \boldsymbol{d} + 2t\boldsymbol{d} \cdot \boldsymbol{l} + \boldsymbol{l} \cdot \boldsymbol{l} - r^2 = 0 \tag{5.5}$$

substitute:

$$\begin{aligned} a &= \boldsymbol{d} \cdot \boldsymbol{d} \\ b &= \boldsymbol{d} \cdot \boldsymbol{l} \\ c &= \boldsymbol{l} \cdot \boldsymbol{l} - r^2 \end{aligned}$$

The equation can be simplified by making d a unit vector, thus giving us $a = 1$. With this assumption being held, an equation in the following form is obtained:

$$at^2 + 2bt + c = 0 \tag{5.6}$$

Using a formula for roots of a quadratic equation t can be found:

$$t = \frac{-2b \pm \sqrt{(2b)^2 - 4c}}{2} \tag{5.7}$$

By carrying out the division on the right side of the formula, a final form of the formula to compute ray-sphere intersection for ray casting is derived:

$$t = -b \pm \sqrt{b^2 - c} \tag{5.8}$$

If the equation does not have solution in real numbers, it means there is no intersection and the fragment is discarded. Otherwise it is simple to compute the position of the intersection from the parameter t. Computing normal at this position is easy, too, because the normal can be obtained by subtracting the sphere origin from the intersection.

Ray Casting of Spherical Triangles

As a spherical triangle is a part of a sphere, the formula for computing line-sphere intersection from the previous section is used to find the intersection with a sphere. The intersection is then clipped using three clip planes.

The biggest difference between the ray casting of spherical triangles and spheres is that the spherical triangle cowers only a small portion of the whole sphere and thus using a cube for ray casting would mean computing many intersections that would be later discarded. For that reason we introduced a different enclosing object.

We have evaluated two possibilities: a prism and a triangular pyramidal frustum, as suggested and implemented by Mgr. Martin Maňák.

Prism Transformation

If the prism is to be used, it has to be transformed from an initial position into a position suitable for ray casting. This is very similar to transforming one triangle into another in 3D.

The transformation needed to change an input prism to arbitrary output prism can be described by a matrix A:

$$\boldsymbol{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

For the purposes of finding the transformation matrix, the prism is seen as a triangle that is moved along the z axis. We will be transforming a triangle:

$$\begin{aligned}\boldsymbol{X_1} &= [x_{1,1}, x_{1,2}, x_{1,3}, 1]^T \\ \boldsymbol{X_2} &= [x_{2,1}, x_{2,2}, x_{2,3}, 1]^T \\ \boldsymbol{X_3} &= [x_{3,1}, x_{3,2}, x_{3,3}, 1]^T\end{aligned}$$

using the matrix A to a new triangle:

$$\begin{aligned}\boldsymbol{Y_1} &= [y_{1,1}, y_{1,2}, y_{1,3}, 1]^T \\ \boldsymbol{Y_2} &= [y_{2,1}, y_{2,2}, y_{2,3}, 1]^T \\ \boldsymbol{Y_3} &= [y_{3,1}, y_{3,2}, y_{3,3}, 1]^T\end{aligned}$$

The transformation can be written as:

$$\begin{aligned}\boldsymbol{AX_1} &= \boldsymbol{Y_1} \\ \boldsymbol{AX_2} &= \boldsymbol{Y_2} \\ \boldsymbol{AX_3} &= \boldsymbol{Y_3}\end{aligned}$$

This means that a system of 12 equations for 12 unknowns has to be solved. Solving this system of equations would be too time-consuming for rendering using shaders. However, the equations can be simplified by carefully selecting the initial triangle as $\boldsymbol{X_1} = [0, 0, z, 1]^T$, $\boldsymbol{X_2} = [1, 0, z, 1]^T$ and $\boldsymbol{X_3} = [0, 1, z, 1]^T$. A prism that meets these requirements can be seen in Figure 5.4. To accommodate the z-coordinate, the destination triangle is modified to include a translation in the direction of a normal $\boldsymbol{n} = [n_1, n_2, n_3]$, i.e. $\boldsymbol{Y_1} = [y_{1,1} + n_1 z, y_{1,2} + n_2 z, y_{1,3} + n_3 z, 1]^T$, similarly for $\boldsymbol{Y_2}$ and $\boldsymbol{Y_3}$.

This change gives us $a_{1,4} = a_{2,4} = a_{3,4} = 1$ directly and it results in a much simpler system of equations:

$$\begin{aligned}
a_{1,3}z \quad\quad\quad\quad &+1 = n_1 z + y_{1,1} \\
a_{1,3}z \quad +a_{1,1} \quad &+1 = n_1 z + y_{2,1} \\
a_{1,3}z \quad +a_{1,2} \quad &+1 = n_1 z + y_{3,1} \\
a_{2,3}z \quad\quad\quad\quad &+1 = n_2 z + y_{1,2} \\
a_{2,3}z \quad +a_{2,1} \quad &+1 = n_2 z + y_{2,2} \\
a_{2,3}z \quad +a_{2,2} \quad &+1 = n_2 z + y_{3,2} \\
a_{3,3}z \quad\quad\quad\quad &+1 = n_3 z + y_{1,3} \\
a_{3,3}z \quad +a_{3,1} \quad &+1 = n_3 z + y_{2,3} \\
a_{3,3}z \quad +a_{3,2} \quad &+1 = n_3 z + y_{3,3}
\end{aligned}$$

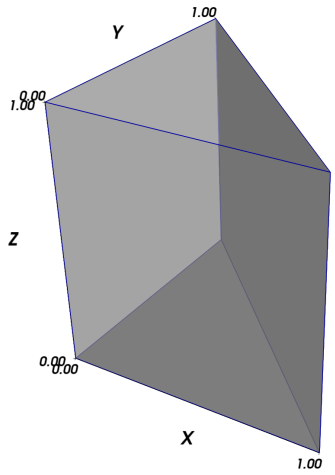


Figure 5.4: A prism that is used for ray casting of spherical triangles.

By solving this system of equations a following definition of A is obtained:

$$\boldsymbol{A} = \begin{pmatrix} y_{2,1} - y_{1,1} & y_{3,1} - y_{1,1} & \frac{n_1\, z + y_{1,1}}{z} - \frac{1}{z} & 1 \\ y_{2,2} - y_{1,2} & y_{3,2} - y_{1,2} & \frac{n_2\, z + y_{1,2}}{z} - \frac{1}{z} & 1 \\ y_{2,3} - y_{1,3} & y_{3,3} - y_{1,3} & \frac{n_3\, z + y_{1,3}}{z} - \frac{1}{z} & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.9}$$

where $z \neq 0$.

This matrix can be used to transform the specified prism into an arbitrary prism in 3D space.

However, this approach brings a new problem – finding a suitable prism that encloses the ray-casted surface. An obvious solution, using the corners of a spherical patch as a prism base, cannot be used because the spherical patch may not fit into such a prism completely. In the worst case, the prism base would have to be large enough to contain an inscribed circle of the three atoms.

Therefore, a solution using a triangular frustum developed by Mgr. Martin Maňák is used. The solution is described in the next section.

Triangular Frustum

The current solution uses a triangular frustum that has been designed by Mgr. Martin Maňák. This method constructs a triangular frustum whose base is defined by the corners of the spherical triangle. The frustum is extruded in the direction of the vectors from the sphere center to the triangle corners and that is big enough to contain the whole triangle.

The method uses the same prism as the previous algorithm and transforms it to a triangular frustum. However, instead of finding a transformation matrix to change the input prism into another prism, a set of conditional transforms is used to transform the prism vertices directly.

The triangle corners are identified using the normalized vectors $\boldsymbol{u_0}$, $\boldsymbol{u_1}$, $\boldsymbol{u_2}$ pointing from the sphere center $\boldsymbol{C}$ to the spherical triangle corners. We call these vectors *corner vectors*. As the distance of the triangle corners from $\boldsymbol{C}$ is equal to the radius r, the position of the base of the frustum can be computed by translating the sphere center in a direction of each corner vector for a distance equal to radius. Therefore, the position of the corners of the spherical triangle, which also form the base of the frustum, can be described as $\boldsymbol{C} + \boldsymbol{u_0}r$, $\boldsymbol{C} + \boldsymbol{u_1}r$, $\boldsymbol{C} + \boldsymbol{u_2}r$.

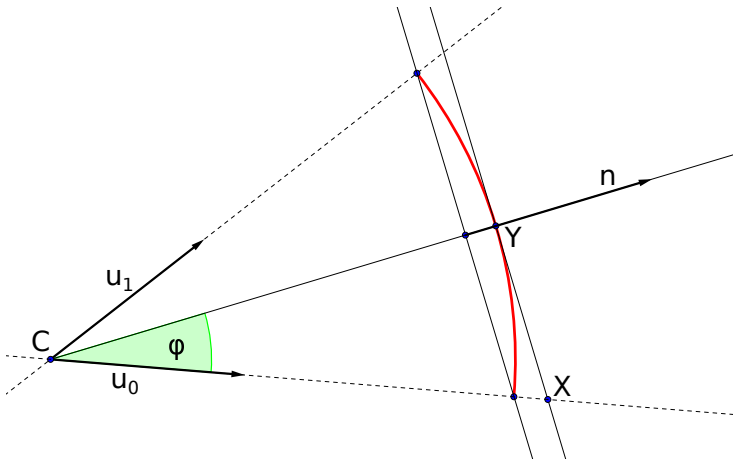


Figure 5.5: Depiction of how a point of a frustum base is found.

The position of the second base can be computed using trigonometry. Figure 5.5 shows a two-dimensional cut of the spherical triangle. To ensure the frustum encloses the whole spherical triangle, the second base is constructed such that it touches the spherical triangle at exactly one point $\boldsymbol{Y}$. It also has to be parallel to the first base with the normal $\boldsymbol{n}$. To find a point of the

second base, the sphere center $\boldsymbol{C}$ is translated in the direction of a corner vector to the plane that meets these requirements. For example, the point $\boldsymbol{X}$ is constructed by translating the sphere center in the direction of the corner vector $\boldsymbol{u_o}$.

To effectively construct the point $\boldsymbol{X}$, the distance $\|\boldsymbol{XC}\|$ by which the vector $\boldsymbol{u_o}$ is multiplied has to be found. The definition of cos can be used:

$$\cos\varphi = \frac{\|\boldsymbol{YC}\|}{\|\boldsymbol{XC}\|} \tag{5.10}$$

$$\|\boldsymbol{XC}\| = \frac{\|\boldsymbol{YC}\|}{\cos\varphi} \tag{5.11}$$

The $\cos\varphi$ can be computed using a dot product:

$$\boldsymbol{n} \cdot \boldsymbol{u_o} = \|\boldsymbol{n}\|\|\boldsymbol{u_o}\|\cos\varphi \tag{5.12}$$

If both $\boldsymbol{n}$ and $\boldsymbol{u_o}$ are normalized, the $\cos\varphi$ can be obtained directly. The distance $\|\boldsymbol{YC}\|$ is known as it is the radius of the sphere.

By applying this method for all three corner vectors, we obtain the points of the second base. An example of a spherical triangle enclosed in a triangular frustum is shown in Figure 5.6.

Ray Casting of Tori

The most complicated part of the visualization is the visualization of tori. Without loss of generality we can assume that the torus is symmetric about z-axis. The results can then be transformed to a desired position using a transformation matrix. The torus is rendered using a cube mesh like it is done when rendering spheres in Section 5.2. Because the torus is symmetric about z-axis, the enclosing object is in fact an axis-aligned bounding box.

A standard implicit equation of torus symmetric about z-axis is:

$$\left(R - \sqrt{x^2 + y^2}\right)^2 + z^2 = r^2 \tag{5.13}$$

Where R is the major radius and r is the minor radius of torus. This equation is used by Krone [21].

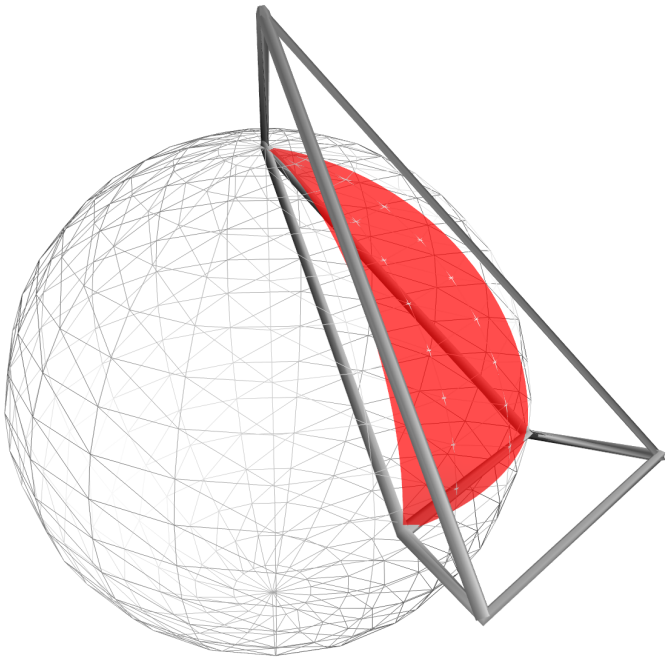


Figure 5.6: A spherical triangle enclosed in a triangular frustum used for ray casting.

Computing a ray-torus intersection is rather complicated (see below), so we try to reduce the search space where the intersection is being searched as much as possible. As numerical methods are used to find the intersection, this includes finding a good initial estimate, too.

The first step in reducing the search space is properly scaling the cube in the vertex shader. To make sure the torus fits into the box, the box is scaled in the xy plane using the major radius. The major radius can be computed as a distance of a probe center from the axis of the torus. Next, the box needs to be scaled along the z axis. To reduce the search space, the box is scaled so that it encloses only the visible part of the torus i.e. a cube that encloses only surface between the tangential points on the two spheres (the green arc in Figure 5.7) rather than the whole torus height. This is done by finding two scales, one being applied to the part of the box that is above the xy plane and one applied to the part below the plane. We will call these scales $zlim_a$ and $zlim_b$.

Computing the scales is based on trigonometry. Figure 5.8 describes the problem using a two-dimensional cut of the torus. Point $\boldsymbol{P}$ is the position of

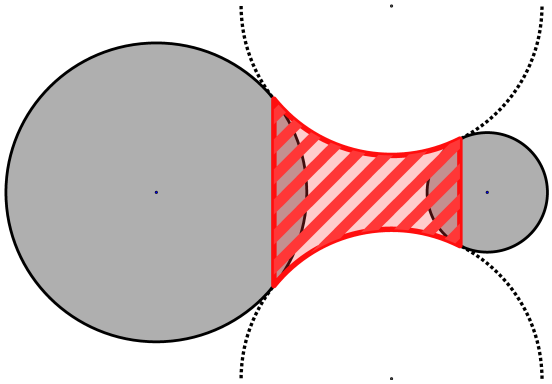


Figure 5.7: A two-dimensional cut of torus. Gray circles represent atoms. The dotted lines outline the whole inner part of a torus. However, the actual surface comprises only of the part highlighted in red and filled with stripes.

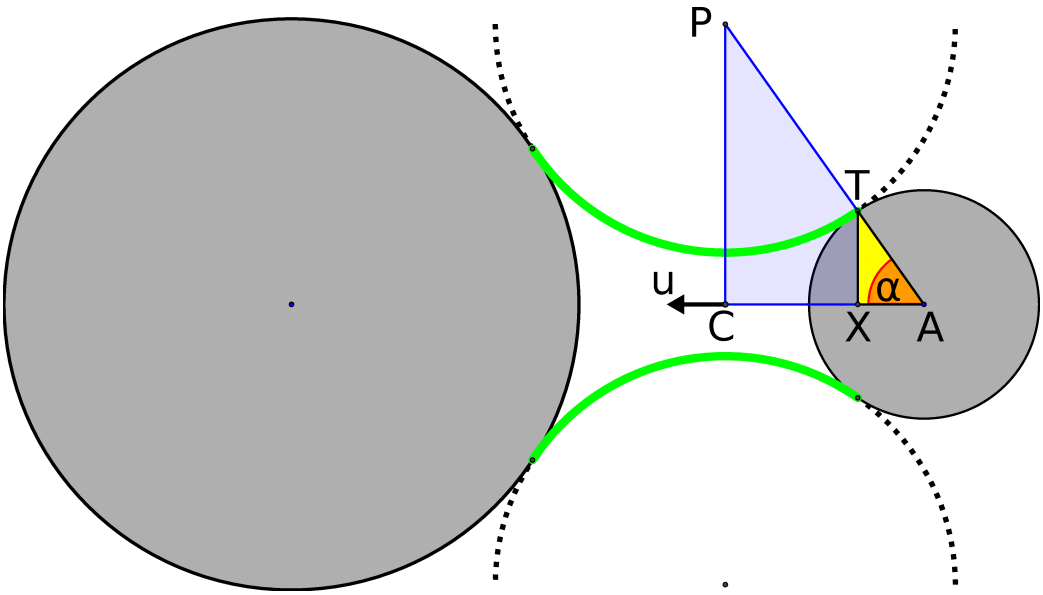


Figure 5.8: Depiction of how the usable length of a torus is found.

the revolving probe. Point $\boldsymbol{C}$ is the point in the center of the torus height. What we want to find is the length $\|\boldsymbol{CX}\|$, which is used as $zlim_a$. The $zlim_b$ is computed in the same way. The length can be computed as:

$$\|\boldsymbol{CX}\| = \|\boldsymbol{AC}\| - \|\boldsymbol{AX}\| \tag{5.14}$$

$\|\boldsymbol{AC}\|$ can be computed using dot product:

$$\boldsymbol{u} \cdot \boldsymbol{AP} = \|\boldsymbol{AP}\|\|\boldsymbol{u}\| \cos\alpha \tag{5.15}$$

where $\boldsymbol{u}$ is a vector aligned to the torus axis. If $\boldsymbol{u}$ is a unit vector, the

equation is simplified to:

$$\boldsymbol{u} \cdot \boldsymbol{AP} = \|\boldsymbol{AP}\| \cos \alpha \tag{5.16}$$

and because:

$$\cos \alpha = \frac{\|\boldsymbol{AC}\|}{\|\boldsymbol{AP}\|} \tag{5.17}$$

the dot product is the length of $\boldsymbol{AC}$.

The length $\|\boldsymbol{AX}\|$ can be computed in the same way as:

$$\|\boldsymbol{AX}\| = \boldsymbol{u} \cdot \boldsymbol{AT} \tag{5.18}$$

The vector $\boldsymbol{AT}$ has the same direction as the vector $\boldsymbol{AP}$ and its length is equal to the sphere radius.

After the bounding box was properly placed and scaled, we can continue with finding a good initial guess for finding intersection and with reducing the search space even more. From now on, everything is carried out in the fragment shader.

The next step is crucial for estimating the initial interval where the roots of the ray-torus intersection equation will be searched. Intersections with a circular frustum whose top and bottom radii are equal to the respective radii of a torus are computed. This gives a good initial estimate for the root finding. Even more importantly it also removes the parts of the top and the bottom where the roots should not be searched to avoid artifacts caused by finding non-existent roots or roots outside the surface.

The idea behind constructing the frustum is as follows: start with an equation of a cylinder symmetric about z-axis:

$$x^2 + y^2 = r^2 \tag{5.19}$$

At any height, the cylinder has the same radius. To create a frustum that has different radii at the top and at the bottom, the radius r^2 is replaced with an interpolation between two radii r_1 and r_2:

$$x^2 + y^2 = r_1^2(1 - z_0) + r_2^2 z_0 \tag{5.20}$$

However, this solution does not take the height of the torus into account yet. To do that, it is necessary to transform the real z-coordinate to a value in the

range $[0, 1]$ using the $zlim_a$ and $zlim_b$ that were used to scale the bounding box:

$$z_0 = \frac{z - zlim_a}{zlim_b - zlim_a} \tag{5.21}$$

To solve the intersection, the parametric equation of a line is substituted for each variable, which results in a quadratic equation:

$$at^2 + bt + c = 0 \tag{5.22}$$

Where a, b and c are as follows:

$$\begin{aligned} a &= (b - a)\ \left({d_y}^2 + {d_x}^2\right) \\ b &= -d_z\,{r_2}^2 + d_z\,{r_1}^2 + (2\,b - 2\,a)\ d_y\,o_y + (2\,b - 2\,a)\ d_x\,o_x \\ c &= (a - o_z)\ {r_2}^2 + (o_z - b)\ {r_1}^2 + (b - a)\ {o_y}^2 + (b - a)\ {o_x}^2 \end{aligned} \tag{5.23}$$

This yields an intersection with an infinitely long circular frustum. Finding an intersection with a frustum with a finite length is just a matter of finding an intersection with a top and bottom planes, which are planes parallel with xy at height $zlim_a$ and $zlim_b$. The result of this step are two parameters t_0 and t_1 where t_0 is the intersection with the front side of the frustum and t_1 with the back side.

At this stage the search space can be reduced by doing clipping that is caused by the probe not being able to roll around the atoms completely. The torus is clipped by two planes at this point, possibly replacing t_0 and t_1 with a parameter that reduces the search space even more. It is required to distinguish two cases: a case when the central angle of the visible part of the torus is convex and a case when the angle is reflex. Both situations are shown in Figure 5.9.

First, a parameter t of the intersection of a ray with each plane is computed. We will label them t_a for an intersection with plane A and t_b for an intersection with plane B. The algorithm slightly differs depending on whether the central angle is convex or reflex. In the convex case, there are four possible configurations of the ray in relation to the clip planes. Figure 5.9 presents example vectors $\boldsymbol{v_0}$, $\boldsymbol{v_1}$, $\boldsymbol{v_2}$, $\boldsymbol{v_3}$ in each possible configuration. We will use these vectors to represent their respective configuration in the algorithm pseudocode. Each configuration can be recognized using a dot product between the vector and a plane. For example, for the vector $\boldsymbol{v_0}$ and vectors $\boldsymbol{a}$, $\boldsymbol{b}$ of plane coefficients the following holds:

$$(\boldsymbol{v_0} \cdot \boldsymbol{a} > 0)\ AND\ (\boldsymbol{v_0} \cdot \boldsymbol{b} > 0) \tag{5.24}$$

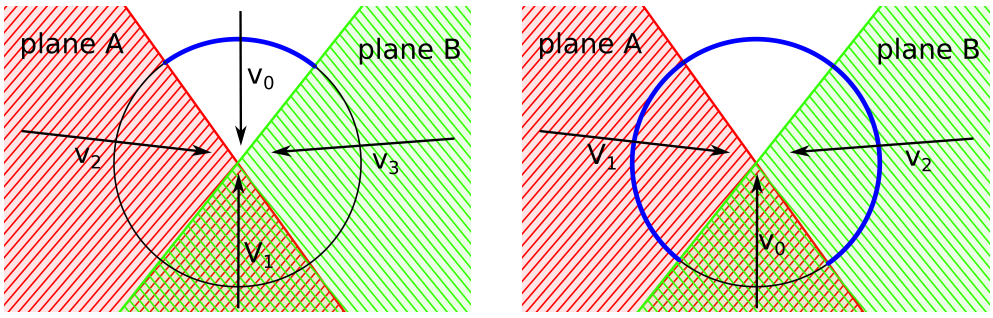


Figure 5.9: A cut of a torus viewed along the Z axis to display clipping. Strong blue outline is the visible part of a torus. Colored half space represents the negative side of each clip plane. The left image presents a torus with a convex central angle. On the right a torus with a reflex central angle can be seen.

The algorithm that adds clipping of the frustum in case the central angle is convex can be written using the following pseudocode:

```
switch (configuration)
  case v_0:
    t_1 = min(t_a, t_b)
  case v_1:
    t_0 = max(t_a, t_b)
  case v_2:
    t_0 = max(t_0, t_a)
    t_1 = min(t_1, t_b)
    if (t_1 < t_0)
      discard
  case v_3:
    t_0 = max(t_0, t_b)
    t_1 = min(t_1, t_a)
    if (t_1 < t_0)
      discard
```

When the central angle is reflex, the algorithm is written as follows:

```
switch (configuration)
  case v_0:
    t_0 = max(t_0, min(t_a, t_b))
  case v_1:
    if (t_b < t_0)
      t0 = t_a
  case v_2:
```

```
if (t_a < t_0)
  t0 = t_b
```

The final step before computing intersections is discarding the fragments where no solution lies. The used algorithm is very simple and fast. However, it may discard even some valid fragments. The algorithm takes parameters t_0 and t_1 obtained in the previous steps and computes their average obtaining $t_{0.5}$. A position is computed for each of these parameters. Each position is then substituted into the equation 5.28.

According to the Bolzano's theorem, if the function has opposite signs on an interval, there must be a root in this interval. Therefore, if there is a change of sign between points t_0, $t_{0.5}$ and t_1, there must be root. If there is no sign change, the fragment is discarded. This gives significant speedup of the computation by discarding the fragments outside of the torus early. It also removes all artifacts caused by a numerical algorithm trying to find a non-existent solution. Again, to reduce the search space, t_0 and t_1 are transformed to the interval where the sign change occur.

However, this method has some drawbacks, too. When the ray intersects both the top and the bottom plane, all points will give the same sign. Still, there may be a root in the inner arch of a torus. Fortunately, this is not such a big problem since for a large part of a surface only the outer part is visible. Even when the inner part is visible, the problem affects only a small portion of the torus either on the bottom or on the top.

Now that the search space was reduced considerably and the initial guesses were found, a ray-torus intersection needs to be computed.

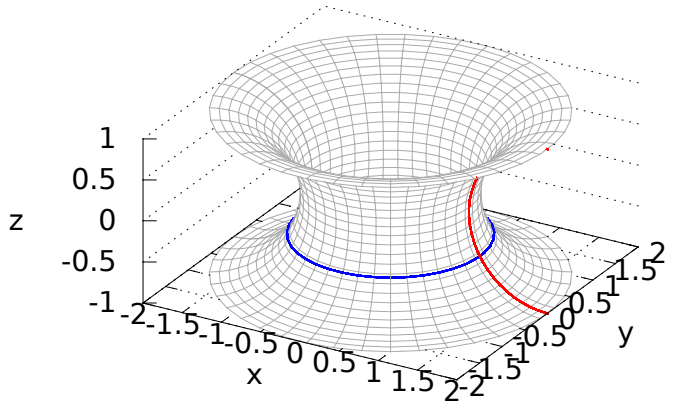


Figure 5.10: An inner part of a torus generated by revolving a semicircle.

Because only the inner part of torus is needed, a slightly modified equation is used in place of a standard implicit equation of a torus. The idea is that the cross section of the inner part of a torus symmetric about z-axis is a circle, whose radius depends on the z-coordinate as illustrated in Figure 5.10. This can be written as:

$$x^2 + y^2 = f(z) \tag{5.25}$$

To find $f(z)$ an equation of a circle in xz plane is solved for x:

$$\begin{aligned} x^2 + z^2 &= r^2 \\ x &= \sqrt{r^2 - z^2} \end{aligned} \tag{5.26}$$

This gives one solution in real numbers that describes a semicircle in a xz plane. To accommodate the major radius, a shift needs to be added to this value, obtaining:

$$f(z) = R - \sqrt{r^2 - z^2} \tag{5.27}$$

Finally $f(z)$ is substituted to the original equation to obtain a final form of the equation of the inner part of a torus:

$$x^2 + y^2 = \left(R - \sqrt{r^2 - z^2}\right)^2 \tag{5.28}$$

The advantage of using this equation instead of the standard implicit torus equation is that it guarantees that any solution is an intersection with the inner part of a torus. With the standard formula the intersections with the inner part of a torus and the intersections with the outer part of a torus need to be distinguished.

For ray casting, it is necessary to compute an intersection of this inner part of a torus with a line. Again, a parametric equation of a line is used:

$$\begin{aligned} x &= o_x + td_x \\ y &= o_y + td_y \\ z &= o_z + td_z \end{aligned} \tag{5.29}$$

By substituting these equations into the equation of the inner part of the torus, a following equation is obtained:

$$(o_x + td_x)^2 + (o_y + td_y)^2 = \left(R - \sqrt{r^2 - (o_z + td_z)^2}\right)^2 \tag{5.30}$$

Solving for t produces a quartic equation of a form

$$a_4 t^4 + a_3 t^3 + a_2 t^2 + a_1 t + a_0 = 0 \tag{5.31}$$

Where:

$$
\begin{aligned}
a0 &= (R^4 - 2r^2R^2 + 2o_z^2R^2 - 2o_y^2R^2 - 2o_x^2R^2 + r^4 - 2o_z^2r^2 - 2o_y^2r^2 - 2o_x^2r^2 \\
&\quad + o_z^4 + 2o_y^2o_z^2 + 2o_x^2o_z^2 + o_y^4 + 2o_x^2o_y^2 + o_x^4) \\
a1 &= (4(d_zo_zR^2 - d_yo_yR^2 - d_xo_xR^2 - d_zo_zr^2 - d_yo_yr^2 - d_xo_xr^2 + d_zo_z^3 \\
&\quad + d_yo_yo_z^2 + d_xo_xo_z^2 + d_zo_y^2o_z + d_zo_x^2o_z + d_yo_y^3 + d_xo_xo_y^2 + d_yo_x^2o_y + d_xo_x^3)) \\
a2 &= (2(d_z^2R^2 - d_y^2R^2 - d_x^2R^2 - d_z^2r^2 - d_y^2r^2 - d_x^2r^2 + 3d_z^2o_z^2 + d_y^2o_z^2 + d_x^2o_z^2 \\
&\quad + 4d_yd_zo_yo_z + 4d_xd_zo_xo_z + d_z^2o_y^2 + 3d_y^2o_y^2 + d_x^2o_y^2 + 4d_xd_yo_xo_y + d_z^2o_x^2 \\
&\quad + d_y^2o_x^2 + 3d_x^2o_x^2)) \\
a3 &= 4(d_z^2 + d_y^2 + d_x^2)(d_zo_z + d_yo_y + d_xo_x) \\
a4 &= (d_z^2 + d_y^2 + d_x^2)^2
\end{aligned}
\tag{5.32}
$$

Although quartic equation can be solved analytically, it is rather difficult and time consuming task, especially when the coefficients are more complicated like in this case. Instead, the equation can be solved numerically. There are various methods to solve the equation numerically, the common choice being bisection method, Newton's method or regula falsi [29]. The initial experiments have shown that the Newton's method fails to converge or converges very slowly for the lower visible part of a torus. While both bisection method and regula falsi gave very similar results in terms of speed and quality, the bisection produced slightly more appealing results.

The bisection uses the parameters t_0 and t_1 found in the previous steps as an initial guess. Our implementation uses 10 iterations to obtain the intersection.

Because the intersection was computed for the torus symmetric about z-axis, it is necessary to transform the intersection using the modelview matrix to the eye space, where shading is performed. The modelview matrix is assembled in the vertex shader, taking the position of generators into account.

6 Filtering

There may be an enormous amount of cavities found in the molecule, with a large part often being of little to no practical use. Thus it is desirable to hide the cavities that do not meet the requirements. The used algorithm allows obtaining additional information about cavities easily, which enables us to filter cavities on the base of various measures. In this chapter, several methods to evaluate cavity properties that can be used for filtering the results are proposed. Methods can be purely geometric, as the methods described in Section 6.1, or they can be based on chemical properties as described in Section 6.2.

6.1 Geometric Methods

The geometric methods do not take the chemical properties and the knowledge about atoms into account. Instead, they rely solely on the cavity geometry, such as its shape or volume. The previous solution implemented only the geometric methods. These are described first. During the course of this work, new geometric methods were developed. These methods are presented later.

Existing Methods

A very important property of a cavity is its volume, because it is easy to understand and gives a good estimate of how big the cavity is. The volume is computed in the following way: first, an axis-aligned bounding box of the Voronoi vertices is obtained and its volume is computed. Then a number of random sampling points are computed in the bounding box. For each point, it is determined whether it hits a probe outside of the cavity surface or not. If the points do not hit a probe outside of the surface, the point is counted as hitting the cavity volume. From the total number of samples and the number of samples that hit the cavity, a probability of a hit can be evaluated. The approximate volume is computed as a multiple of the hit probability and a volume of the bounding box.

Another property that was previously implemented by Mgr. Martin Maňák

is a complexity based on the number of spherical triangles. It is a simple measure of the number of spherical triangles that are part of the cavity surface. A high number usually means that the cavity is large and that it has complicated surface. Small cavities and cavities with a simple surface tend to have a lower number of spherical triangles in their surface.

The next two properties are the minimum and the maximum probe. The minimum probe is the radius of a sphere when the cavity becomes part of the outer void. This tells us when the cavity becomes accessible from the outer space. The maximum probe is the radius of a sphere that fits into the largest cell in the cavity quasi-triangulation. It is applicable when the radius of the largest spherical space in the cavity is needed.

There is also an "aggressive" filtering of cavities. This filtering is based on a maximum probe and the number of Voronoi vertices in a cavity. The cavities are sorted according to the maximal probe radius and the number of vertices. The filter returns only a selected percentage of cavities from each sorted list, while avoiding duplicates if a cavity fulfills both the requirement for the maximal probe size and the number of vertices.

New Methods

As a part of this work, new methods for evaluating geometric properties of a cavity were introduced to further extend the knowledge about cavities and to allow more informed filtering of the cavities. These properties are focused mostly on describing the shape of cavity. They are: fractal dimension, convexity and branching. Each of these will be described in more detail in the following sections.

Fractal Dimension

One of the geometric methods is a method that uses fractal dimension to measure the roughness of the cavity surface. While, strictly speaking, the cavity surface is not a fractal due to missing small scale features, it still has some degree of self-similarity at higher scales, making it possible to estimate its fractal dimension.

The idea behind fractal dimension is to enumerate how a fractal pattern

changes at various scales. The term fractal dimension usually refers to a Hausdorff dimension [14] or to a box-counting dimension [14] Both dimensions express the self-similarity at various scales.

The Hausdorff dimension is based on a concept of covering the space F by sets of increasingly smaller diameter. As the diameter decreases, the number of sets covering the space increases. To find the dimension, we first define a δ-cover of F as a cover of space with sets of radius at most δ. Then we define Hausdorff measure as:

$$H^s_\delta(F) = \inf\{\sum_{i=1}^{\infty} r_i^s\} \tag{6.1}$$

where r_i is a δ-cover of F. In other words we seek to minimize the sum of the r_i^s. As previously mentioned, the number of sets increases with decreasing δ. The Hausdorff dimension is defined as a limit when δ approaches 0:

$$H^s(F) = \lim_{\delta \to 0} H^s_\delta(F) \tag{6.2}$$

The box-counting dimension approaches the problem by using boxes instead. First, $N_\delta(F)$ is defined as the smallest number of sets with diameter at most δ that cover F. Then a box-counting dimension is given by:

$$dim_b(F) = \lim_{\delta \to 0} \frac{\log N_\delta(F)}{-\log \delta} \tag{6.3}$$

To compute the box-counting dimension, F is placed onto a uniform grid and a number of boxes that covers F is counted [35]. The box-counting dimension can be seen as a ratio how the required number of boxes changes as $\delta \to 0$.

It can be seen that the box-counting dimension is much easier to compute. For the Hausdorff dimension it is necessary to compute spheres to cover the examined object. However, for box counting dimension a uniform grid can be used. Also, determining whether a box covers an object is much simpler than doing the same for a sphere. For this reason, the box-counting dimension is more commonly used.

When computed for a cavity, the box-counting dimension can be seen as describing the roughness of a cavity surface, or its complexity. The higher the dimension is, the more complex the cavity is.

To simplify the computation and to make things faster, a cavity is approximated using spheres in the Voronoi vertices. The algorithm works as follows: first, a uniform grid with a defined spacing is created. Second, the algorithm

goes through all cavity vertices and marks each block in the grid that intersects the sphere defined by the vertex. Then a number of marked blocks is counted. This algorithm is repeated using several block sizes. Next the results are portrayed as points. The box-counting dimension is the slope of this line. To find the slope, a linear least squares method is used.

Convexity

We define the convexity property as a ratio of a convex hull volume and a cavity volume. For the purpose of computing volume the atoms are replaced by points and a cavity is represented as a set of tetrahedra. This allows using simpler algorithms to compute the volumes.

The convex hull is computed using the QuickHull3D library [4] that computes a convex hull using the QuickHull algorithm [7]. Once the convex hull is computed, three random vertices are selected from the hull and their position is averaged to obtain a middle point x. This point is certain to lie inside the convex hull. It is used to compute a volume of the hull by iterating over all triangular faces of the hull. For each face a, b, c, a tetrahedron a, b, c, x is computed and its volume is computed using the equation 6.4. The volume of the hull is the sum of volumes of all tetrahedra.

$$V = \frac{1}{6}||(a - x) \cdot ((b - x) \times (c - d))|| \tag{6.4}$$

To compute the volume of a cavity, we loop over all tetrahedra of a quasi-triangulation and compute the sum of their volumes.

We define convexity as a ratio:

$$\frac{V_{triangulation}}{V_{hull}} \tag{6.5}$$

It yields a number between 0 and 1, where 1 is a cavity that is perfectly convex. The closer the value is to zero, the more inconvexities are present in a cavity. The convexity can be used to quantify how difficult it would be to slip probe through the cavity. There is a higher chance that the probe gets stuck in a more complicated cavity with many inconvexities rather than in a convex cavity.

Branching

The branching property is based on the algorithm that finds the most substantial subgraph of a graph. The algorithm is used to reduce a graph describing a cavity leaving only its most prominent features. From this graph, a "branching number" is computed as a sum of the number of additional branches of nodes with a degree higher than two. This number describes a number of the most important paths in a cavity graph, thus it can be used to evaluate how complicated the cavity is.

The input of the algorithm for finding the most substantial subgraph is a graph and a metric that defines the distance between two nodes in the graph. The output of the algorithm is a graph. The algorithm iteratively removes insignificant nodes from the graph.

The algorithm begins by finding all graph endpoints, i.e. nodes of degree one. These nodes are assigned a distance 0 and added to a list of start nodes. Each endpoint is then used as a starting node for a depth-first search. The search stops when it discovers a node with degree > 2. A node with degree > 2 is considered a "branching point" of a graph and it is where insignificant branches are removed. A distance from the endpoint to the branching point is stored in the branching point and the branching point is added to a list of future start nodes. It should be pointed out that when doing the depth-first search, the branching point should not be marked as processed once it is found. That is because the branching point may be discovered from another endpoint, too.

After all endpoints are processed, the algorithm starts pruning the graph to remove insignificant branches. To prune the branches, the algorithm iterates over the list of future start nodes (i.e. the current branching points). For each node, it removes the paths whose distance to the endpoints is smaller than a user-specified value, while leaving at least one path with the assigned distance intact. If the removal reduced the degree of the point to less than two, the longest two paths are kept. If a path has no computed distance yet, it is not considered for removal.

The algorithm continues by replacing the list of start points with the list of future start points. The algorithm then tries to find new branching points using the same algorithm. The algorithm stops when there are no branching points found.

An example of how the the algorithm works step-by-step is given in Figure 6.1.

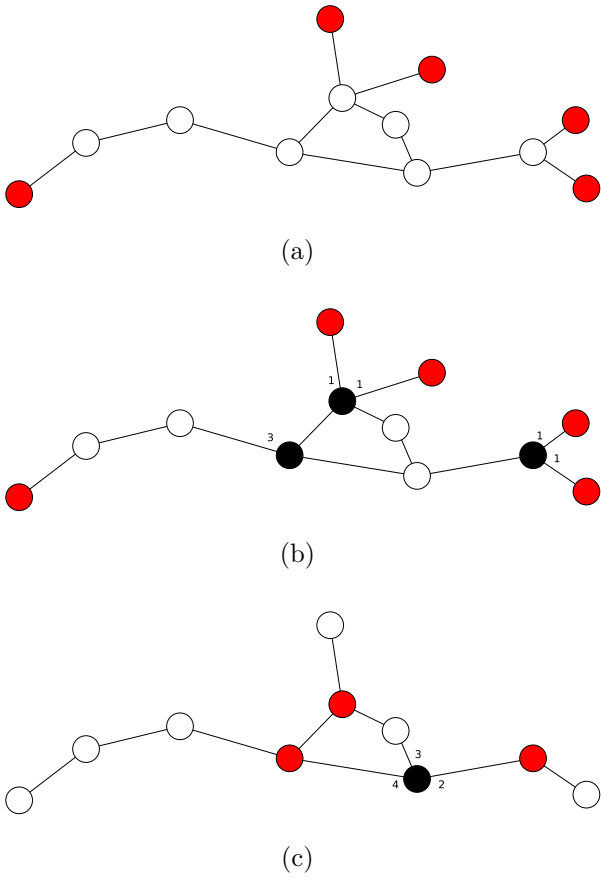


Figure 6.1: The example of removing short branches. For the illustrative purposes length of an edge is 1. The branches shorter than 2 are removed. (a) The initial graph with endpoints marked with red. (b) First iteration of the algorithm, branching points are black, with the incoming edges assigned with a distance to the endpoint on the path from that edge. (c) The short branches are removed, keeping at least one branch intact. A new branching point is computed and distances are assigned.

For the purposes of computing the branching number, the algorithm to reduce the cavity is used with a metric that is an Euclidean distance of the Voronoi vertices. The branching number is computed by adding up the degree - 2 of all nodes with degree > 2.

It would be possible to introduce other cavity measures using the algorithm to find the most substantial subgraph by changing the metric, or by using a different algorithm to measure the resulting subgraph. An option could be the use of this algorithm, e.g. to measure the longest path, possibly taking the path smoothness into account.

6.2 Atom-based Filtering

Apart from the filtering methods based on geometric properties, it can be useful to filter cavities based on their chemical properties, too. This is important if there is a requirement for some chemical interaction to occur in a cavity. For this purpose, a filtering method based on residues adjacent to a cavity was implemented in this work.

The input of this method is a user selectable list of residues. The cavities are filtered by testing all atoms forming each cavity if they are part of a residue selected by the user. The cavities that do not contain an atom with any selected residue are removed.

7 Results

The algorithms described in this work were implemented in a plugin for Caver [9], a software tool for protein analysis and visualization. In the first section, the related functionality of the plugin will be covered. In the second section, the comparison with the previous solution will be discussed. Finally, a few usage examples of the plugin will be given.

7.1 Caver Plugin

The algorithms presented in this work were implemented in a Caver plugin for finding cavities. This plugin was developed by Mgr. Martin Maňák. The plugin was modified to find approximate cavities using the Delaunay triangulation, while allowing the user to select cavities for the precise computation using the additively-weighted Voronoi diagram. The rendering using ray casting was added in addition to the previous solution. The recent version uses ray casting as a default setting. Also, the existing filtering options were extended with the new filtering options described in this work.

Cavities Using the Delaunay Triangulation

Implementing the approximate cavities involved the following changes: adding a computation of the Delaunay triangulation, modifying the code to allow the cavities to be recomputed using the additively-weighted Voronoi diagram, and modifying the user interface to reflect the changes.

We decided to use CGAL [2] to compute the Delaunay triangulation rather than implementing our own solution. The reason to choose CGAL over other solutions is that the implementation of the Delaunay triangulation in CGAL is proved to be a well-tested and very flexible, yet fast implementation. The downside of choosing CGAL is that CGAL is written in C++, while the plugin is written using Java. To solve this, a wrapper using Java Native Interface was implemented to allow using the CGAL's implementation from Java. Also in this wrapper, the bottleneck computation is included, compared to the previous solution which uses a separate module to compute bottlenecks. This is because the CGAL's implementation of Delaunay trian-

gulation provides additional information which includes most of the data for bottleneck computation.

To allow easier interoperability with the existing algorithms, the wrapper provides the same interface as the existing module for computing the additively-weighted Voronoi diagram from the previous solution. Thanks to this, the code for finding cavities in a diagram could be left mostly unchanged, allowing the change between the precise computation and the approximate computation being done by simply changing the module which does the preprocessing.

Changes to the code were also needed to allow improving the approximate cavities using the additively-weighted Voronoi diagram to find a precise cavity shape. Probably the biggest challenge lied in changing the existing code that assumed that the triangulation and bottlenecks computed in the preprocessing stage are valid for the whole molecule (i.e. all cavities are computed using the same triangulation and the same bottlenecks). To allow an improvement of individual cavities, it had to be changed, because each cavity can be computed using a different triangulation and different bottlenecks.

To improve an individual cavity, we iterate over all Voronoi vertices of a cavity, while adding their generating atoms to a HashSet. The HashSet is used to avoid duplicates caused by the fact that many generators are shared by multiple vertices. The atoms are used to run the algorithm using the additively-weighted Voronoi diagram to find precise cavities.

The user interface reflects these changes in a way that it allows the user select interesting approximate cavities and request their precise computation. It also presents the user with an information whether a cavity is only approximate or if it was computed precisely with the additively-weighted diagram. An example of the user interface will be given in Section 7.4.

Ray Casting

The ray casting is implemented using GLSL shaders and it is integrated with the existing code. The implementation uses instancing, implemented by Mgr. Maňák to reduce the number of data transmitted to a video card. For all shapes, a buffer containing a bounding object is generated. This bounding object is a simple primitive modeled using a triangular mesh. Depending on the shape it is either a cube with coordinates ranging from $[-1, -1, -1]$ to

$[1, 1, 1]$ for spheres and tori, or a triangular prism aligned with with a base $[0, 0, 0]$, $[1, 0, 0]$ and $[0, 1, 0]$ and height from -1 to 1 for spherical triangles. The shaders are passed the necessary information needed to transform the default object to a bounding object required for the computation.

Generally, the vertex shaders are used to translate the bounding object into its position and to scale and rotate it to make sure it encloses the rendered shape. Also in the vertex shader, the rays from camera to a vertex are computed to be used for ray casting. The fragment shader computes an intersection between a rendered shape and the ray, including clipping. The intersection is used to compute the lighting and correct depth.

The implementation of the ray casting in shaders closely reflects the process described in the theoretical part in Section 5.2.

Filtering

The current Caver plugin allows attaching various information to each cavity by so-called *Cavity Properties*. The information can be used to sort and filter cavities in a list of all cavities. All new geometric methods presented in Chapter 6 are implemented as Cavity Properties. To use the existing infrastructure, each cavity property must implement the required interfaces and it must be correctly annotated. When done properly, the property is automatically picked up and made accessible from the user interface. The user interface shows each available property. The user can use this newly acquired knowledge about cavities to manually hide or show the cavities.

The atom-based filtering is implemented separately because of different requirements on the user interface. The implementation is hooked to the code right after the cavity-finding. When the cavities are found, they are checked against the selected residues and filtered before they are even visualized. To make this possible, it is necessary to keep mapping between vertex generators and the atoms. To filter a cavity, it is necessary to find out which residue each atom belongs to. The caver provides the necessary functions. All the cavity residues are checked against the user selected set of residues. To improve the speed of the check, the user selected residues are stored in a hash set to quickly identify whether the user selected the tested residue.

7.2 Comparison

In this section, a comparison with the previous solution will be presented. The comparison comprises several parts. The first part focuses on the time required to find cavities. In the next part, the ray casting is compared to the previous solution using triangular meshes. Finally, selected cavities are used to present the cavity measures.

The system used for experiments was a PC with Core i7 920 CPU (four cores at 2.7GHz with Hyper-Threading), 12 GB RAM and GPU nVidia GeForce 9600 GT 512MB. The used operating system was Arch Linux 64bit running Oracle JDK 7u55.

When measuring times, the first few runs of each algorithm were omitted until the times become more stable. This is to ensure the code was loaded into memory, and more importantly, that the measurements were not affected by JIT compiler optimizing the code. The optimization is otherwise exhibited by shortening times between successive runs.

Finding Cavities

This section presents a comparison between the previous solution and the proposed solution in terms of speed of finding cavities.

First, we have evaluated the preprocessing on a variety of molecules with a different number of atoms. All molecules used for testing are available from the RSCB Protein Data Bank [3] under the public domain with the exception of the molecule 70S RF2 [20] that is available from the Center for Molecular Biology of RNA [5]. We refer to all molecules using their PDB IDs.

The measured times required for preprocessing using the Delaunay triangulation are presented in Table 7.1. The preprocessing times for the method using the additively-weighted Voronoi diagram are in Table 7.2. The tables contain two times – a time required to build a triangulation from the input and a total time required for the preprocessing, including the computation of bottlenecks.

It can be seen that the time required for preprocessing is several times shorter when using Delaunay triangulation. An interesting result is that there is

much higher disparity between the time needed to compute the triangulation and the time needed for the rest of the preprocessing despite the fact that the implementation of the Delaunay triangulation in CGAL already includes some of the steps needed to compute the bottlenecks. This is likely due to the overhead caused by the need to pass data from the C++ structures to Java objects.

It can also be seen that the required time does not depend solely on the number of atoms. For example, the times for molecule 1FFK were very short despite its high number of atoms. This is probably because this molecule is very sparse and the atoms are grouped into several smaller clusters.

Molecule (# atoms)		**Execution time** [ms]					**Average** [ms]
1CQW	*triangulation*	20	19	19	20	24	20
(2 358)	*total*	105	102	102	103	108	**104**
1FFK	*triangulation*	28	28	27	27	27	27
(3 656)	*total*	156	160	159	158	159	**158**
1CW2	*triangulation*	39	39	39	38	39	39
(4 926)	*total*	228	218	220	214	217	**219**
3O58	*triangulation*	194	192	189	195	189	192
(23 719)	*total*	1 149	1 120	1 118	1 145	1 092	**1 125**
3OH5	*triangulation*	211	211	209	210	222	213
(25 880)	*total*	1 250	1 211	1 278	1 269	1 278	**1 257**
3WOD	*triangulation*	237	239	239	239	245	240
(28 953)	*total*	1 399	1 446	1 443	1 403	1 423	**1 423**
2VKZ	*triangulation*	711	721	720	714	711	715
(85 830)	*total*	4 127	4 311	4 182	4 390	4 172	**4 236**
70S RF2	*triangulation*	2 616	2 544	2 510	2 577	2 557	2 561
(298 820)	*total*	16 656	15 982	14 870	15 605	15 509	**15 724**

Table 7.1: Times required for preprocessing using the Delaunay triangulation. The values in the triangulation row describe the time needed to compute the triangulation of atoms only. The row total contains the total time required to build a triangulation and to compute bottlenecks.

In the next comparison we compared a total time required by each method to find a precise cavity. If we want a precise cavity, the time needed to improve the cavity constitutes to the total required time. The example of times required to find a precise shape of selected cavities using both approaches (i.e. finding approximate cavities and improving a cavity and finding cavities

Molecule (# atoms)		**Execution time** [ms]					**Average** [ms]
1CQW	*triangulation*	430	456	462	442	459	450
(2 358)	*total*	531	552	555	531	552	**544**
1FFK	*triangulation*	280	287	278	278	288	282
(3 656)	*total*	345	353	348	344	353	**349**
1CW2	*triangulation*	199	200	198	203	186	197
(4 926)	*total*	242	250	241	245	228	**241**
3O58	*triangulation*	2 488	2 467	2 510	2 484	2 443	2 478
(23 719)	*total*	2 940	2 911	2 971	2 925	2 913	**2 932**
3OH5	*triangulation*	2 654	2 773	2 787	2 738	2 714	2 733
(25 880)	*total*	3 154	3 268	3 300	3 224	3 202	**3 230**
3WOD	*triangulation*	3 219	3 394	3 464	3 360	3 387	3 365
(28 953)	*total*	3 794	3 972	4 039	3 936	3 971	**3 942**
2VKZ	*triangulation*	13 912	14 058	13 806	13 423	13 782	13 796
(85 830)	*total*	15 863	16 059	15 778	15 418	15 700	**15 764**
70S RF2	*triangulation*	46 453	47 903	44 157	48 906	46 960	46 876
(298 820)	*total*	53 389	54 960	51 095	55 482	53 899	**53 765**

Table 7.2: Times required for preprocessing using the additively-weighted Voronoi diagram. The values in the triangulation row describe the time needed to compute the triangulation of atoms only. The row total contains the total time required to build a triangulation and to compute bottlenecks.

directly with the additively-weighted Voronoi diagram) is found Table 7.3. It can be seen that the difference is negligible for smaller molecules, but as the molecule size increases, the total time becomes considerably lower when using approximate cavities with the improvement step. However, improving large number of cavities may become ineffective if that meant that the additively-weighted Voronoi diagram had to be computed for a larger portion of the molecule.

Finally, we compared the volume of some selected cavities to identify how much the approximate cavities and the precise cavities differ. The results are presented in Table 7.4. The results show that the approximate cavities have larger volume as expected. This means that when approximate cavities are used, it must be taken into account that the true cavity is likely smaller and it may not meet requirements anymore.

Molecule	Approximate	Precise
1CW2 (1.4 Å)	282	559
1AKD (1.4 Å)	202	355
1AOI (3.0 Å)	346	680
2VKZ (4.2 Å)	6 302	14 776
70S RF2 (8.0 Å)	15 558	57 925

Table 7.3: The total time in [ms] required to find a precise shape of the largest cavities. The probe size is specified next to each molecule ID.

Molecule	Approximate	Precise	Ratio
1CW2 (1.4 Å)	1 664	1 126	1.48
1AKD (1.4 Å)	2 041	1 371	1.49
1AOI (3.0 Å)	3 699	3 306	1.12
70S RF2 (8.0 Å)	30 553	30 114	1.01

Table 7.4: The comparison of volumes of the largest cavities found with a probe size specified under each molecule ID. The volumes were computed using 1000000 samples.

Visualization

To compare the visualization speed, we measured the frames per second (FPS) when rendering a rotating model of a molecule.

Rather than measuring FPS when visualizing surface of inner cavities, we visualized the outer surface of a whole molecule. The reason to do so is that it is less dependent on the structure of a molecule. Small molecule may contain more cavities than a large molecule when the same probe is used. Also, the outer surface has much larger area than the surface of cavities, thus it is better for stress testing.

For all measurements, we used a probe of size 1.4 Å, as it is a very common choice. Both the old method described in Section 4.3 and the ray casting described in Section 5.2 were tested on molecules of various sizes to show how the methods scale for increasingly large data. The FPS were measured over a period of time (several minutes) to obtain a larger set of samples. The samples were averaged and the results are presented in Table 7.5.

Molecule	Old method	Ray casting
1CQW	5.255	17.933
1CW2	2.983	10.940
3WOD	0.741	6.552
1FFK	2.874	9.354
2VKZ	0.348	0.935
3O58	0.761	2.020
70S RF2	0.161	0.325

Table 7.5: The frames per second when rendering a surface of selected molecules using a probe 1.4 Å.

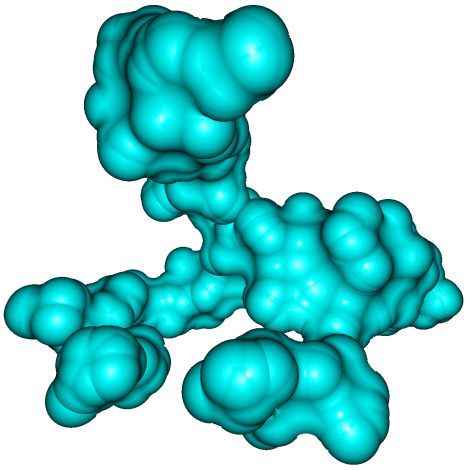
7.3 Filtering

New filtering methods were introduced in this work. In this section some real-world examples of cavities are presented while providing the measured data that can be used for filtering. The user can use the measured properties to sort cavities in a list of all cavities and to manually filter cavities.

We have selected three different cavities to demonstrate the new geometric methods that can be used to measure cavities for filtering. The cavities are shown together with the evaluation of their properties as implemented in Caver plugin in the Figure 7.1, 7.2 and 7.3. The volume is added to keep a sense of scale.

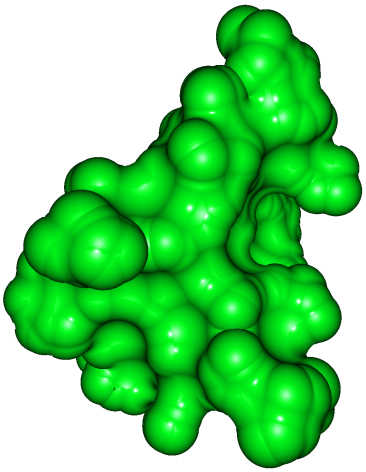
Because of its large, tentacle-like spurs, the cavity in Figure 7.1 has a very low convexity. Its shape also causes the fractal dimension to be relatively high, since the surface varies a lot on different scales. Also the branching measure is high, indicating that the cavity has a complicated shape. While the cavity in Figure 7.2 has a relatively high convexity compared to the first one, it still has a high fractal dimension and the branching measure because of the surface roughness at a smaller scale. Figure 7.3 shows a simple, round-shape cavity. According to that, it has a very high convexity, very low branching and low fractal dimension.

The results show that the selected properties can be used to sort and filter cavities based on their shape. The convexity is a more global measure and can be used to filter cavities with lots of bends where a molecule can easily get stuck. The fractal dimension describes finer details of the surface, which affects the probability of a contact between the surface and other molecules [6]. The branching is affected by both finer details and a general shape of a cavity.



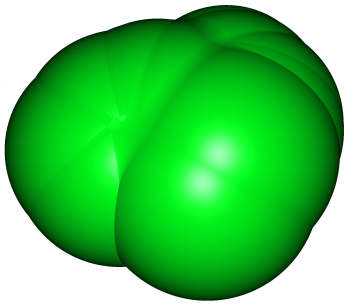
Parameter	Value
Volume	1404
Convexity	0.43
Fractal Dimension	2.17
Branching	352

Figure 7.1: The largest precise cavity in a molecule 1CW2 for a probe 1.3 Å.



Parameter	Value
Volume	1371
Convexity	0.64
Fractal Dimension	2.18
Branching	380

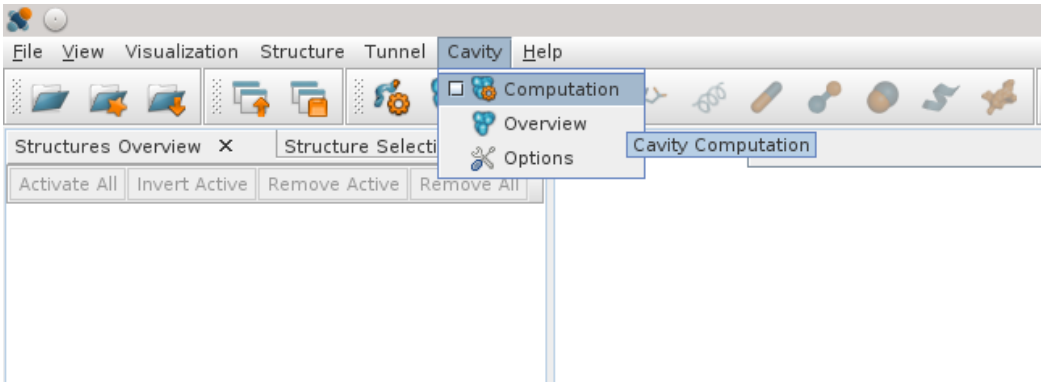
Figure 7.2: The largest precise cavity in a molecule 1AKD for a probe 1.4 Å.



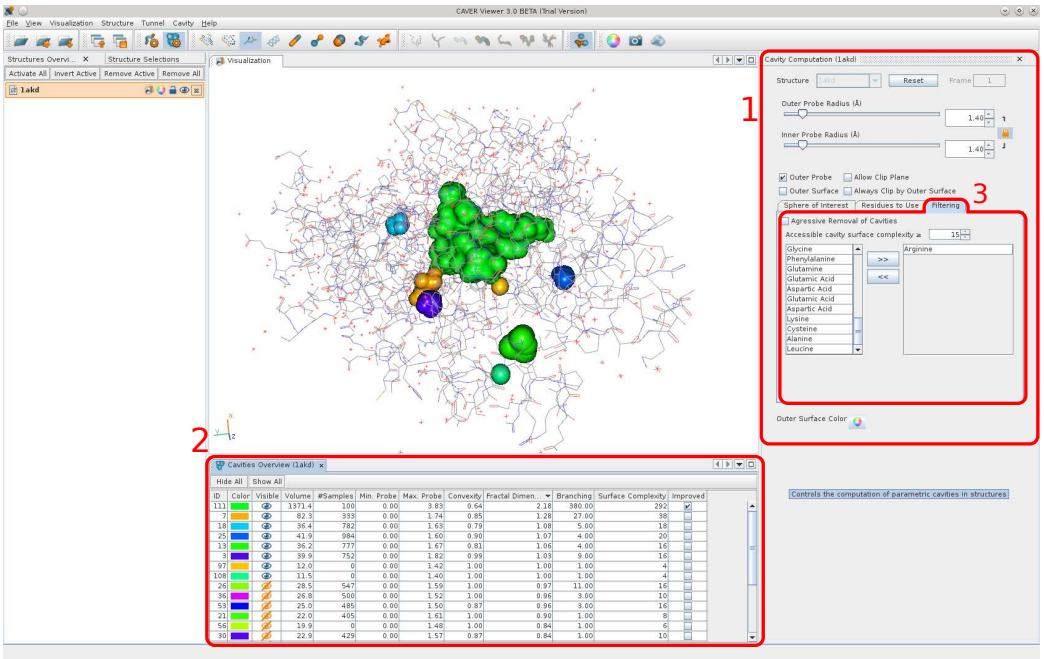
Parameter	Value
Volume	400
Convexity	0.88
Fractal Dimension	1.7
Branching	6

Figure 7.3: The largest precise cavity in a molecule 1AKD for a probe 3.4 Å.

7.4 User Interface



A menu to enable the user interface of the plugin to find cavities. "Computation" shows the interface to find the cavities. "Overview" shows a list of all cavities with their properties. The overview can be used to filter cavities.



Caver with a loaded molecule 1AKD. The control interface for cavity finding is on the right (1). The list of all cavities ("Overview") is on the bottom (2). The cavities are sorted according to the "Fractal Dimension" and only the ones with dimension > 1 are shown. The first cavity in the list has been improved, which is depicted in the column "Improved." To improve a cavity, right click on a selected cavity, or cavities, in the list and select improve. The atom based filtering is accessible from the panel (3). Only the cavities with Arginine are shown.

8 Conclusion

In this work, we have presented a modification of an algorithm using the additively-weighted Voronoi diagram to find cavities that allows processing large molecules. The modification uses the Delaunay triangulation to find approximate cavities. The shape of an approximate cavity can be improved by a modified algorithm using the additively-weighted Voronoi diagram that considers only the atoms forming the cavity.

To allow visualization of cavities in large molecular models a solution using the solvent-excluded surface of cavities was developed. This solution uses ray casting to render the basic shapes forming the surface.

Because the user is usually interested in a subset of cavities, we have provided new means to measure cavity properties that can be used for filtering cavities based on both purely geometrical approach and on residues forming the cavity.

The experiments show that computing approximate cavities and then improving only selected cavities leads to an improvement over the previous solution in the terms of speed. Also, the visualization using ray casting proved to be of very high quality, while being several times faster than a method using triangular mesh. Finally, we have visually evaluated the newly introduced cavity properties for the filtering.

Bibliography

[1] awVoronoi: A library for computing 3D additively weighted Voronoi diagrams. `http://awvoronoi.sourceforge.net/`.

[2] CGAL, Computational Geometry Algorithms Library. `http://www.cgal.org`.

[3] RCSB Protein Data Bank. `http://www.rcsb.org/pdb/`.

[4] QuickHull3D: A Robust 3D Convex Hull Algorithm in Java. `https://www.cs.ubc.ca/~lloyd/java/quickhull3d.html`.

[5] Center for Molecular Biology of RNA. `http://rna.ucsc.edu/`.

[6] BANERJI, A. – NAVARE, C. Fractal nature of protein surface roughness: a note on quantification of change of surface roughness in active sites, before and after binding. *Journal of Molecular Recognition*. 2013, Vol. 26, Vol. 5, pp. 201–214. ISSN 1099-1352. DOI 10.1002/jmr.2264.

[7] BARBER, C. B. – DOBKIN, D. P. – HUHDANPAA, H. The Quickhull Algorithm for Convex Hulls. *ACM Trans. Math. Softw.* December 1996, Vol. 22, Vol. 4, pp. 469–483. ISSN 0098-3500. DOI 10.1145/235815.235821.

[8] BERNSTEIN, H. J. – CRAIG, P. A. Efficient molecular surface rendering by linear-time pseudo-Gaussian approximation to Lee–Richards surfaces (*PGALRS*). *Journal of Applied Crystallography*. Apr 2010, Vol. 43, Vol. 2, pp. 356–361. DOI 10.1107/S0021889809054326.

[9] CHOVANCOVÁ, E. et al. CAVER 3.0: A Tool for the Analysis of Transport Pathways in Dynamic Protein Structures. *Pathways in Dynamic Protein Structures, PLoS Computational Biology 8: e1002708*. 2012.

[10] CONNOLLY, M. L. Analytical molecular surface calculation. *Journal of Applied Crystallography*. Oct 1983, Vol. 16, Vol. 5, pp. 548–558. DOI 10.1107/S0021889883010985.

[11] EDELSBRUNNER, H. The union of balls and its dual shape. *Discrete & Computational Geometry*. December 1995, Vol. 13, pp. 415–440. ISSN 0179-5376.

[12] EDELSBRUNNER, H. – MüCKE, E. P. Three-dimensional alpha shapes. *ACM Transactions on Graphics*. January 1994, Vol. 13, pp. 43–72. ISSN 0730-0301.

[13] EXNER, T. et al. Identifification of Substrate Channels and Protein Cavities. *Molecular modeling annual*. 1998, Vol. 4, Vol. 10, pp. 340–343. ISSN 0949-183X. DOI 10.1007/S008940050091.

[14] FALCONER, K. *Fractal Geometry: Mathematical Foundations and Applications*. John Wiley & Sons, Inc., 2003. ISBN 978-0470848623.

[15] HENDLICH, M. – RIPPMANN, F. – BARNICKEL, G. LIGSITE: automatic and efficient detection of potential small molecule-binding sites in proteins. *Journal of Molecular Graphics and Modelling*. 1997, Vol. 15, Vol. 6, pp. 359 – 363. ISSN 1093-3263. DOI 10.1016/S1093-3263(98)00002-3.

[16] JIRKOVSKÝ, L. Finding Cavities in a Molecule. In *Proceedings of the 16th Central European Seminar on Computer Graphics*, pp. 33–40, Smolenice, Slovakia, 2012. ISBN 978-3-9502533-4-4.

[17] KESSENICH, J. – BALDWIN, D. – ROST, R. The OpenGL® Shading Language: Language Version: 1.20. Technical report, 3Dlabs, Inc. Ltd.

[18] KIM, D.-S. – CHOB, Y. – KIM, D. Euclidean Voronoi diagram of 3D balls and its computation via tracing edges. *Computer-Aided Design*. 2005, Vol. 37, pp. 1412–1424.

[19] KIM, D.-S. et al. Quasi-triangulation and interworld data structure in three dimensions. *Computer-Aided Design*. 2006, Vol. 38, Vol. 7, pp. 808–819.

[20] KOROSTELEV, A. et al. Crystal structure of a translation termination complex formed with release factor RF2. *Proceedings of the National Academy of Sciences*. 2008, Vol. 105, Vol. 50, pp. 19684–19689. DOI 10.1073/pnas.0810953105.

[21] KRONE, M. – BIDMON, K. – ERTL, T. Interactive Visualization of Molecular Surface Dynamics. *Visualization and Computer Graphics, IEEE Transactions on*. Nov 2009, Vol. 15, Vol. 6, pp. 1391–1398. ISSN 1077-2626. DOI 10.1109/TVCG.2009.157.

[22] LASKOWSKI, R. A. SURFNET: A program for visualizing molecular surfaces, cavities, and intermolecular interactions. *Journal of Molecular Graphics*. 1995, Vol. 13, Vol. 5, pp. 323 – 330. ISSN 0263-7855. DOI 10.1016/0263-7855(95)00073-9.

[23] LEE, B. – RICHARDS, F. The interpretation of protein structures: Estimation of static accessibility. *Journal of Molecular Biology*. 1971, Vol. 55, Vol. 3, pp. 379 – IN4. ISSN 0022-2836. DOI 10.1016/0022-2836(71)90324-X.

[24] LEVITT, D. G. – BANASZAK, L. J. POCKET: A Computer Graphics Method for Identifying and Displaying Protein Cavities and Their Surrounding Amino Acids. *J. Mol. Graph.* December 1992, Vol. 10, Vol. 4, pp. 229–234. ISSN 0263-7855. DOI 10.1016/0263-7855(92)80074-N.

[25] LIANG, J. et al. Analytical shape computation of macromolecules: II. Inaccessible cavities in proteins. *PROTEINS: Structure, Function, and Genetics*. October 1998, Vol. 33, Vol. 1, pp. 18–29.

[26] LIANG, J. – WOODWARD, C. – EDELSBRUNNER, H. Anatomy of protein pockets and cavities: Measurement of binding site geometry and implications for ligand design. *The Protein Society*. September 1998, Vol. 7, Vol. 9, pp. 1884–1897.

[27] MANAK, M. – JIRKOVSKY, L. – KOLINGEROVA, I. Computation and Visualization of Molecular Surfaces for Various Probe Sizes. Article in preparation, 2014.

[28] OKABE, A. et al. *Spatial tessellations: Concepts and applications of Voronoi diagrams*. John Wiley & Sons, Inc., 2nd edition, 2000. ISBN 978-0-471-98635-5.

[29] PRESS, W. H. et al. *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing*. New York, NY, USA : Cambridge University Press, 1992. ISBN 0-521-43108-5.

[30] RICHARDS, F. M. Areas, Volumes, Packing, and Protein Structure. *Annual Review of Biophysics and Bioengineering*. 1977, Vol. 6, Vol. 1, pp. 151–176. DOI 10.1146/annurev.bb.06.060177.001055. PMID: 326146.

[31] ROTH, S. D. Ray casting for modeling solids. *Computer Graphics and Image Processing*. 1982, Vol. 18, Vol. 2, pp. 109 – 144. ISSN 0146-664X. DOI 10.1016/0146-664X(82)90169-1.

[32] SANNER, M. – OLSON, A. J. – SPEHNER, J. C. Reduced Surface: an Efficient Way to Compute Molecular Surfaces. *Biopolymers*. 1996, Vol. 38, Vol. 3, pp. 305 – 320.

[33] Schrödinger, LLC. The PyMOL Molecular Graphics System, Version 1.7.0.0. 2014.

[34] SONAVANE, S. – CHAKRABARTI, P. Cavities and Atomic Packing in Protein Structures and Interfaces. *PLoS Computational Biology*. September 2008, Vol. 4, Vol. 9, pp. e1000188.

[35] SUZUKI, M. T. A Three Dimensional Box Counting Method for Measuring Fractal Dimensions of 3D Models. In *Proceedings of the Eleventh IASTED International Conference on Internet and Multimedia Systems and Applications*, IMSA '07, pp. 42–47, Anaheim, CA, USA, 2007. ACTA Press. ISBN 978-0-88986-678-2.

[36] TRIPATHI, A. – KELLOGG, G. E. A novel and efficient tool for locating and characterizing protein cavities and binding sites. *Proteins: Structure, Function, and Bioinformatics*. 2010, Vol. 78, Vol. 4, pp. 825–842. ISSN 1097-0134. DOI 10.1002/prot.22608.

[37] WILLIAMS, M. A. – GOODFELLOW, J. M. – THORNTON, J. M. Buried waters and internal cavities in monomeric proteins. *Protein Science*. August 1994, Vol. 3, Vol. 8, pp. 1224–1235.