Rendering Molecular Surfaces as Implicit Surfaces on GPUs

MSc Thesis

Sérgio Emanuel Duarte Dias

Rendering Molecular Surfaces as Implicit Surfaces on GPUs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE AND ENGINEERING
(Engenharia Informática)

by

Sérgio Emanuel Duarte Dias
natural of Lisbon, Portugal

Computer Graphics and Multimedia Group
Department of Computer Science and Engineering
Univeristy of Beira Interior
Covilhã, Portugal
www.di.ubi.pt

Rendering Molecular Surfaces as Implicit Surfaces on GPUs

Autor: Sérgio Emanuel Duarte Dias
Número de Aluno: m1419
Email: sergioduartedias@sapo.pt

Resumo

A modelação computacional de superfícies moleculares permite-nos extrair informações importantes sobre a interacção entre moléculas, assim como sobre as áreas e os volumes de tais moléculas. Vários algoritmos têm sido desenvolvidos para a representação e renderização de superfícies moleculares. Contudo, todos estes algoritmos padecem da falta de eficiência no que respeita ao tempo que levam a visualizar superfícies moleculares, o que se deve ao facto destes algoritmos terem sido desenhados para a arquitectura tradicional baseada na CPU. Uma solução possível para resolver este problema está no uso de sistemas de computação paralela, que eram até há relativamente pouco tempo sistemas extremamente caros. Mas com o aparecimento da nova geração de GPUs programáveis de baixo custo, e com grande capacidade de processamento em paralelo, abriu-se uma janela de oportunidades para resolver este problema. Assim nesta tese é apresentado um algoritmo desenvolvido para GPUs por forma aumentar a velocidade de renderização de superfícies moleculares. Além do desenvolvimento do algoritmo também foi realizado um estudo comparativo do desempenho entre uma versão sequencial (CPU) do algoritmo e a versão em paralelo (GPU) do algoritmo para renderizar superfícies de connolly, bem como superfícies de van der Waals.

Orientador: Prof. Dr. Abel Gomes, DI , UBI

Rendering Molecular Surfaces as Implicit Surfaces on GPUs

Author: Sérgio Emanuel Duarte Dias
Student Number: m1419
Email: sergioduartedias@sapo.pt

Abstract

Modeling molecular surfaces enables us to extract useful information about interactions with other molecules, as well as measurements of molecular areas and volumes. Many types of algorithms have been developed to represent and rendering molecular surfaces. However, these algorithms have questionable time performance in the visualization of molecular surfaces because they are usually designed to run CPU. A possible solution to resolve this problem is the use of parallel computing, but parallel computing systems are in general very expensive. Fortunately, the appearance of the new generation of low-cost programmable GPUs with massive computational power can, in principle, solve this problem. So, in this thesis we present a GPU-based algorithm to speed up the rendering of molecular surfaces. Besides we carry out a study that compares a sequential version (CPU) to a parallel version (GPU) of well-know Marching Cubes (MC) algorithm to render Connolly surface, as well as van der Waals surfaces.

Supervisor: Prof. Dr. Abel Gomes, DI , UBI

Preface

I would like to thank my supervisor Prof. Dr. Abel Gomes for his rigorous guidance to every step of my thesis. Also, by every help he gave in the several doubts I had in computer graphics, CUDA, and molecular visualization. To my parents for their support, specially the advices that (among many other things) allowed me to conclude my degree. To my friends Amador for the discussions we had on both our theses, VBOs, and CUDA programming in general, and Alex Cardoso and Ricardo S. Alexandre for annoying me and Amador to a point of making us want to work just not to put up with them. To other friends for the help, support, and encouragement during the period of writing up this thesis.

Sérgio Emanuel Duarte Dias
Covilhã, Portugal
August 28, 2009

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

The field of molecular biology has received a growing attention from computer scientists in the last few years. This field deals with the prediction of 3D structures of molecules and their functionalities. From the structure of a molecule, several important properties may be obtained, namely the behavior of living organisms, enzyme catalysis, among other properties. However, obtaining this structure using traditional methods (i.e., non-computational methods) is complex, slow, and expensive. To address this problem, computational methods started to be used to determine the structure of molecules. However, these methods have revealed computationally heavy in terms of processing time. To alleviate this processing problem, scientists have recently adopted GPU-based parallel architectures to visualize molecules and molecular complexes.

There are several models to visualize molecules, namely: ball-and-stick, CPK space-filling, and now the so-called molecular surfaces. The most used representation is the Connolly surface. This representation allows for inference of various biological properties of a molecule, and the computation of mass properties (i.e. area and volume) that can be easily determined. It is true that there already are some CPU-based visualization systems for molecules (i.e. Chimera), but they only allow the visualization of molecules individually, and without the dynamics of molecular complexes consisting of two or more molecules.

However, until recently, only expensive parallel architectures had the capacity of visualizing molecular complexes. With the appearance of the new generation of low-cost GPUs with massive computation power this problem no longer exists. It is, in principle, enough to form a cluster of NVIDIA GPUs cards to get the performance of a supercomputer, with the advantage of being scalable. It is this massive power that we intend to explore in this thesis in order to represent and visualize molecules, in particular the Connolly surfaces, using for that a low cost PC with a single NVIDIA GPU.

1.1 Motivation

The principal challenge in molecular modeling is the massive number of computing facilities that are needed to visualize biomolecules and molecular complexes, as well as their dynamics in real-time. However, with the appearance of the new generation of low-cost

GPUs with massive computation power, the problem of rendering molecular surfaces in real-time can been, in principle, solved without using supercomputers or conventional parallel computer systems. So, the principal motivation behind this thesis is the possibility of modeling and rendering molecules on the GPU. Also, since the beginning of this work, there was the will of learning a bit more about the emergent GPU technology using CUDA.

1.2 The Problem Statement

The growing of interest in the field of molecular biology results from the relevance of the study of the geometry and topology of molecular structures to better understand the molecular behavior. Visualization programs that give such information already exist (i.e. Chimera). However, because these programs are CPU-based, most of them are too slow to visualize of complexes molecules with good quality. The reason for this arises from the heavy mathematical calculations underlying the modeling of molecular complexes. Another reason that accentuates this problem is the use of visualization algorithms that are very expensive in terms of processing. Both problems contribute for the increase of the total time needed to model and render molecular surfaces. Therefore, to resolve this problem, it is necessary to have high performance systems that are capable of processing massive data in parallel. Speeding up the processing and visualization time of complex molecules is the principal problem of current molecular visualization programs.

1.3 Contributions

This thesis describes the design and implementation of a CUDA-based algorithm that uses the inverse squared function (ISF) to locally approximate the Connolly surface of a molecule. Most molecular visualization programs have been designed to run on the CPU, and only a few ones run on the GPU. However, as far as we know, there is only one to render molecular surfaces using CUDA (i.e. VMD).

Let us then enumerate the contributions of this thesis:

- CUDA-based MC algorithm to run on GPU in the context of molecular modeling.
- The van der Waals surface approximation obtained from the Connolly surface using a Newton Corrector.
- The use of a simple blending function (i.e. inverse squared function) involving a single mathematical operation (division) is a little, but important, contribution of this thesis, since it allows to represent Connolly surfaces quickly on the CPU.

Besides, we have carried out a comparative study of rendering times between CPU-based and GPU-based implementation of the same algorithm (i.e., Marching Cubes algorithm), using three different blending functions (Blinn function, soft object function, and inverse squared function). This study is useful to help us to realize which is the most adequate function to locally approximate molecular surfaces.

1.4 Thesis Organization

Chapter 1 is the current chapter. It introduces the research context that has led to the writing of this thesis, including the motivation, the main problem, and the contributions for the advance of knowledge.

Chapter 2 is an overview of the several molecular surfaces existing in the literature. A description of each molecular surface is given to the reader. Also, here it is given information about the work related to the different triangulation algorithms for molecular surfaces, and their visualization.

Chapter 3 is the core of this thesis. It describes the fast CUDA-based triangulation algorithm of molecular surfaces. In essence, it is a parallel GPU-based marching cubes algorithm.

Chapter 4 carries out a time performance analysis and comparison between CPU and GPU-based marching cubes algorithms.

Chapter 5 summarises the main results obtained during the master research program that has led to this dissertation, and suggests possible directions for future work.

1.5 Target Audience

The target audience of this thesis is not only the communities of computer graphics, and geometry computing, but also the community of computational molecular biology, for whom we have designed our algorithm.

Chapter 2

Implicit Molecular Surfaces: The State-of-the-Art

Visualizing molecules in 3D provides some useful information about biological processes (e.g. protein docking). To do this, it is necessary to collect information about the position, and the type of atoms that compound the molecule. This is done using X-ray crystallography, which is a technique based on the diffraction patterns of a X-ray irradiation in a crystal molecule. Passing irradiation through the crystal causes the scattering of the X-rays deflected by the electrons of the molecule. The result is then recorded in a appropriate plate that provides information about the position of each atom. Fortunately, these molecules, and their constituent atoms, can be now retrieved as a PDB file from the PDB repository or from other repositories easily accessible via Internet.

With this information, we are able to visualize molecular surfaces using adequate triangulation algorithms. This chapter just reviews the most important triangulation algorithms and techniques to render molecular surfaces in the implicit form.

2.1 Molecular Surface Types

A molecule is a set of atoms. Each atom has a nucleus (consisting of protons and neutrons) whose positive charge attracts the nearby electrons. This electron cloud surrounds the atomic nucleus. This electron cloud provides us relevant information, namely the atom size and the forces between the charges of atoms (i.e. electrical field). These forces are referred in the scientific literature as electrical field intensity, which decreases with the distance from the center of the atom. By summing up the electrical field intensities f_i of all atoms, we get the overall electrical field intensity F of the molecule. That is,

$$F = \sum_{i=1}^{N} f_i$$

Thus, for a given threshold of the electrical field intensity, we obtain an isosurface that approximates the molecular surface. By varying the value of this threshold we are able to approximate three well-known types of surfaces: van der Waals surface [32], SAS (Solvent Accessible Surface) [36, 47], and Connolly Surface [22].

2.1.1 van der Waals Surface

In this model, each atom is represented as a sphere. The sphere radius is given by the van der Waals radius of the atom [32].

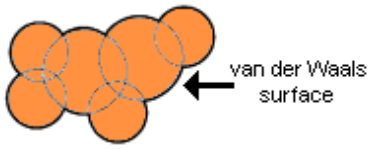

Figure 2.1: Van Der Waals Surface.

As illustrated in Figure 2.1, a van der Waals surface is defined as the boundary of the union of the spherical atoms that compound the molecule. Despite some inaccuracies found in the distribution of electron density, this model is widely used because of the important geometric properties that can be calculated out of it (i.e. areas and volumes) [54].

2.1.2 Lee-Richards Surface

The 3D structure of a molecule provides us with biological functional properties, as needed to understand how a molecule interacts with other molecules (e.g. ligand-substrate relation). The importance of the molecular structure was recognized in 1971 by Lee and Richards [36, 47], who developed the solvent accessible surface model (SAS), also called Lee-Richards surface.

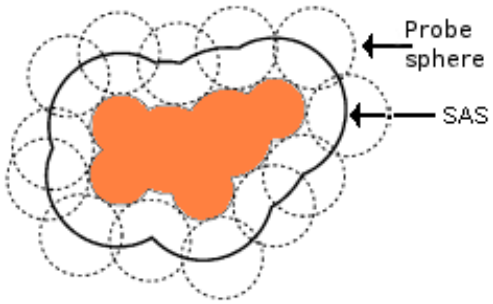

Figure 2.2: Solvent Accessible Surface.

As shown in Figure 2.2, the SAS is obtained by rolling a probe sphere (typically the water molecule) on the atoms of the molecule, being the surface generated by the corresponding sweeping of the center of the probe. Therefore, the resulting molecular surface is essentially an inflated Van Der Waals surface, in which the atomic radii are inflated by the probe radius.

2.1.3 Connolly Surface

The solvent accessible surface (SAS) has many crevices and sharp corners that can be eliminated using a modified sweeping procedure of the probe over the atoms of the molecule. Instead of sweeping the center of the probe sphere while the probe rolls on the molecule, we use surface patches of both atoms and probe sphere to generate a surface, called Connolly surface. Using this alternative method, a smoother surface is obtained without sharp corners and crevices, as it can be seen in the Figure 2.3.

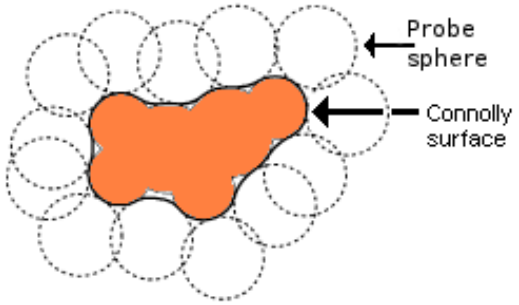

Figure 2.3: Connolly Surface.

Note that, the Connolly surface also results from rolling a probe sphere on the atoms of a molecule. But, now, it is compound from surface patches of two types:

- contact surface patches (from atoms), and
- reentrant surface patches (from the probe)

Contact surface patches are convex surface patches of the atom spheres that are in contact with the probe sphere, i.e., the part of the van der Waals surface of the atoms that are accessible to the probe sphere [23]. The reentrant patches are concave and saddle-shaped toroidal patches. The saddle-shape patches are generated from the probe surface, but they are generated when the probe sphere rolls in contact with two atoms simultaneously. Finally, the concave patches are also generated from the probe surface, when the probe sphere rolls in contact with three atoms simultaneously, as shown in Figure 2.4. To this new way of obtaining a molecular surface was given the name of Connolly surface, after Michael Connolly [23, 22].

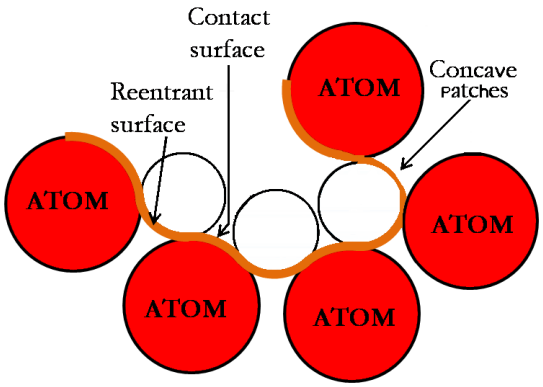

Figure 2.4: Surface patches from Connolly surface.

2.2 Triangulation Methods for Molecular Surfaces

To visualize the previously described types of molecular surfaces (i.e. van der Waals surface, Lee-Richards surface, Connolly surface) there are mainly four methods, namely: Voronoi tessellation, Delaunay triangulation, alpha shapes, and Marching Cubes.

2.2.1 Voronoi Tessellation(Voronoi diagram)

Given a set S of points, in the Euclidean space, the corresponding Voronoi tessellation can be obtained by partitioning the Euclidean space into convex regions, called Voronoi cells or Dirichlet domains, that satisfy two conditions:

- each region includes only one input point.
- the frontier of each region is defined by hyperplanes equidistant from every two closest input points.

That is, given two closest input points, the corresponding hyperplane is defined by the equidistant perpendicular bisector of the line segment that connects them, as illustrated in Figure 2.5.

Richards pointed out two problems in using Voronoi tessellation [47]:

- *Lack of atomic diversity*. The Voronoi tessellation treats all atoms as equivalent, independently of their chemical nature.
- *Lack of neighbor atoms*. Atoms exposed to a solvent on the protein surface has a small number of neighbors. As illustrated in Figure 2.5 the atoms in the frontier of the molecule are approximated by unbounded polyhedra or cells, which are meaningless volumes.

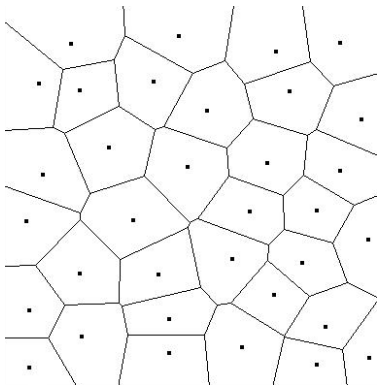

Figure 2.5: Voronoi Diagram.

The first problem has to do with the fact that all atoms of a molecule are considered to have the same size, what is meaningless from the chemistry point of view. However, different atoms have distinct radii. As a geometric consequence, the bisection planes of Voronoi tessellation are not necessarily equidistant from two neighbor atom centers. A bisection hyperplane is only equidistant from two atom centers when both atoms have the same radius. But using this space partioning is not physically accurate, because the atoms of a chemical compound have different sizes. This means that the bisection hyperplane between two atoms no longer cuts the space at the midpoint of their centers. Such a cutting point lies somewhere between the two atom centers in order to make the representation of the molecular surface physically more accurate. There are two main methods to re-position the bisection hyperplane: the method B (Richards [47]) and the radical-plane method (Gellatly [28]).

Let us focus on the method B. This method applies to both covalent and non-covalent bounds. When method B is applied to covalent bounds the bisection hyperplane between the atoms is placed at a position from the first atom given by the following distance:

$$d = \frac{DR}{(R+r)} \tag{2.1}$$

where D is the distance between the atoms, R radius of the first atom, and r radius of the second atom; for non-covalent bonded atoms, the method B uses the following distance:

$$d = R + (D - R - r)/2 \tag{2.2}$$

A second problem with Voronoi tessellation for molecules has to do with the difficulty in constructing closed polyhedra around the surface atoms, which leads to ambiguities in computing there volumes and densities (see an example in Quillin and Matthews [46]). Amongst the possible solutions to overcome the second problem, we find the sphere union representation [40] and the Delaunay triangulation [45]. The Delaunay triangulation is illustrated in Figure 2.6 (on the right-hand side).

Recall that the first work using Voronoi tessellation in molecular modeling was done by Richards in 1974 [47], where an analysis of the atomic packing inside macromolecules was performed. Other works on Voronoi tessellation in molecular modeling are due to Finney [28], Harpaz [33], Gerstein [29], Pontius [44], Nadassy [42], and Tsai [52].

2.2.2 Delaunay Triangulation

In 2D, a Delaunay triangulation can be obtained by connecting the (input) center points of neighbor Voronoi cells, as illustrated in Figure 2.6.

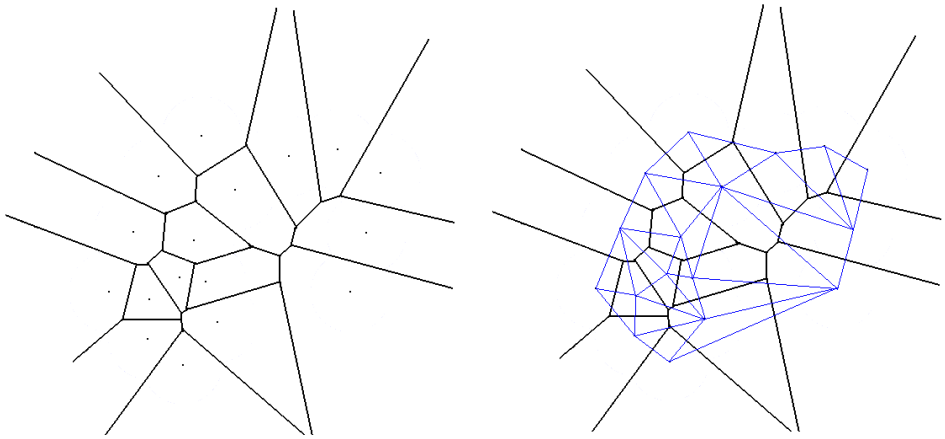

Figure 2.6: A set of points and its Voronoi diagram Right (left): the corresponding Delaunay triangulation (right).

The Delaunay triangulation is formed by joining all neighboring points (i.e. atom centers), where "neighboring" means pairs of points whose Voronoi cells share an edge in 2D (or a face in 3D). As shown in Figure 2.6, the cells of a 2D Delaunay triangulation are triangles. The Delaunay triangulation is also the dual graph of the Voronoi tessellation [26, 30].

The Delaunay triangulation outputs the convex hull of the given points or atom centers, which results in a very rough approximation to a molecule. As a consequence, the Delaunay tessellation cannot represent cavities and holes of molecules. To fix this problem, we can use a weighted Delaunay triangulation [49] or a restricted Delaunay triangulation [20]. Another possible solution for this problem is using alpha shapes [12, 21].

2.2.3 Alpha Shapes

Alpha shapes are also define from a set P of points, which are atom centers in molecular modeling. For a given value of $\alpha \in [0,\infty)$, *alpha balls* represent atom spheres of varying radius α (Figure 2.7. The triangulation of these atom spheres start off when the atom spheres start intersecting. According to Edelsbrunner and Mucke [26], the corresponding triangulation is called *alpha complex* of P, and is the Delaunay triangulation of P *restricted*

to the alpha spheres. For $\alpha = 0$, the alpha complex is exactly the set P, and for sufficiently large α, the alpha complex is the Delaunay triangulation $D(P)$ of P.

Thus, an alpha shape is a family of spherical geometries that are generated from a set of unorganized points, which allows us to find out a family of triangulations for a set of points by varying the α parameter. The main difficulty in using alpha shapes is to determine the correct value of α to represent a triangulation of a molecule without eliminating cavities and holes [26, 18]. For example, the value of α in Figure 2.7-e is incorrect because it removes the cavity shown in Figure 2.7-d.

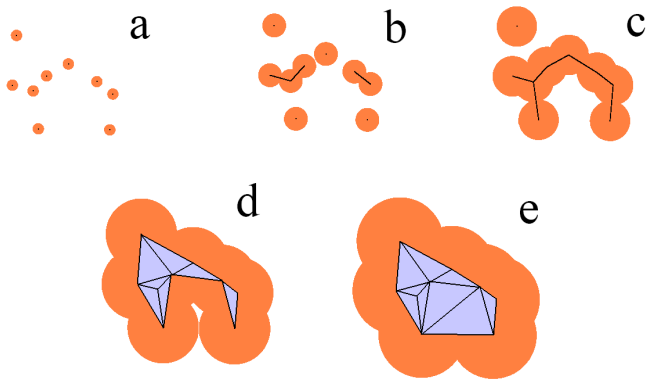

Figure 2.7: alpha shapes for growing values of α.

The use of alpha shapes to molecular modeling is due to Edelsbrunner [26, 25], from which he also introduce the skin surface [24]. These surfaces have a rich combinational structure, and provide a smooth alternative to the Lee-Richards surface. Cheng [19] maintained an approximation triangulation of a deforming skin surface that changes over time.

2.2.4 Marching Cubes

The original marching cubes algorithm was created and presented in 1987 by Lorensen and Cline. It is a very popular isosurface rendering algorithm [38]. The method partitions the bounding box containing the surface into cubes, i.e., a grid of cubes (Figure: 2.8). The function that describes the surface is then evaluated at each vertex.

If the function takes on a value equal or less than the isovalue at a given vertex, this vertex is assigned a bit 0; otherwise, the corresponding bit is set to 1. Note that each cube is associated to a 8-bit mask or index, 1 bit per vertex (Figure 2.9). This process tells us which vertices are inside the surface and those which are outside, as needed to determine the surface topology within each cube. Because we have a single bit for each vertex, and each cube has 8 vertices, we end up having $2^8 = 256$ possible shape configurations of the surface inside a cube. Thus, we can use an 8-bit mask or index to store the shape configuration of the surface inside each cube.

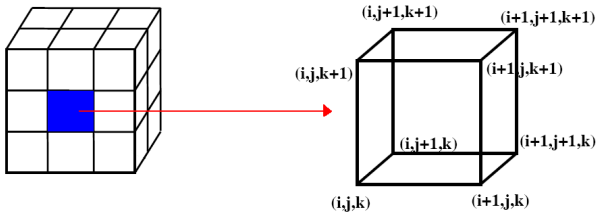

Figure 2.8: Grid of voxels.

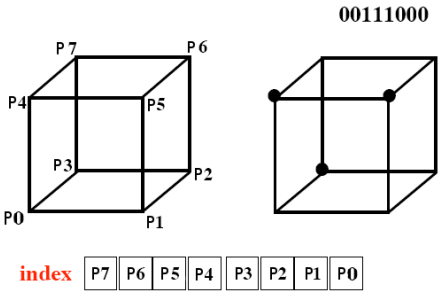

Figure 2.9: Cube Numbering.

Each one of the 256 shape configurations is encoded by an 8-bit index Figure 2.9, and corresponds to a list of edges, where each edge intersects the isosurface. This is illustrated in Figure 2.10, where the index 00111000 determines which are the transverse edges, at a point, called mesh vertex.

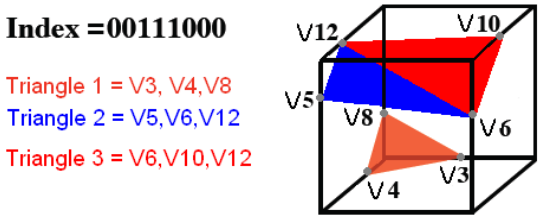

Figure 2.10: List of Edges.

With the list of transverse edges of each cube, the algorithm goes on to find the intersecting vertex location, along each transverse edge using linear interpolation. For this purpose, we use the following formula:

$$P = P_a + \frac{P_b - P_a}{f(P_b) - f(P_a)}(T - f(P_a)) \tag{2.3}$$

where P_a and P_b the endpoints of the transverse edge where, $f(P_a)$ and $f(P_b)$ are the function values on the previous endpoints, where P is the interpolated, intersecting point on triangulation vertex and T is the threshold.

Terminated the interpolation procedure, it remains to create the corresponding triangles for each cube, and calculate the corresponding normal vectors for shading purposes.

One of the first algorithms to use marching cubes for molecular surfaces is due to Weiden [53]. This is an improved MC algorithm that uses logarithmic interpolation, and avoids artificial gaps and other inconsistencies. After this work, the MC algorithm was used to visualize molecular surfaces, using texture mapping [50, 51]. In Merelli [41] and Agostini [11] a molecular visualization program was developed, using an parallel version of marching cubes.

2.2.5 Other Methods

There are other methods to visualize molecular surfaces. An example is due to Bajaj [14], who proposed a NURBS approximation of the Lee and Richards molecular surfaces. In Bajaj [15] work, methods to maintain molecular surfaces for varying solvent radii were presented. Bajaj [13] also introduced a compressed volumetric representation of molecular surfaces. Holst [35] proposed an alternative solution to visualize molecular surfaces, based in subdivision schemes. These schemes are used to generate tetrahedral meshes for molecular structures, from the solution of the Poisson-Boltzmann equation. Methods based on Gaussian functions have been used to construct density maps [16, 31, 39], from which implicit solvation models are approximated as isocontours [37, 27]. Also through algebraic spline model a method was developed to approximate of a molecular surface [56].

2.2.6 Molecular Surfaces Visualization Programs

It is worth remembering that there are several CPU-based programs to visualize molecular surfaces, namely:

- *BALLView*: a program that allows to visualize ball-and-stick models, molecular surfaces, or ribbon models [1].
- *ChemApplet*: a Java applet for viewing sample molecular structures [2].
- *ChemCraft*: a graphical program for visualization of molecules (i.e. molecular surfaces, ball-and-stick models) [3].
- *Chimera*: an interactive molecular modeling system, that has many features related with the possibility of knowing properties of molecular surfaces [4].
- *Jmol*: a molecular viewer written in Java, that allows to visualize molecular surfaces and ball-and-stick models [5].
- *RasMol*: a program for molecular graphics visualization [7].

- *Viewmol*: an open source molecular graphics visualization program for computational chemistry [9].
- *VMD*: a visualization and analysis program of biological systems, such as proteins [10].

These programs put in evidence the state of the art of molecular modeling from a pratical point of view. They work as a good starting point for geometric modeling research of molecular surfaces.

2.3 Summary

In this chapter, we have presented the different types of known molecular surfaces (i.e. van der Waals surface, Lee-Richards surface, and Connolly surface). Also in this chapter, we have presented the state-of-the-art of triangulation methods to render molecular surfaces. Recall that this chapter does not address parametric formulations of molecular surfaces. The focus is on implicit molecular surfaces and their space partitionings.

Chapter 3

Fast GPU-based Triangulation of Molecular Surfaces

A GPU-based triangulation algorithm to render molecular surfaces is described in this chapter, as well as its implementation. In essence, it is a parallel version of the Marching Cubes (MC) that runs on the GPU using the CUDA API from NVIDIA.

3.1 GPU Programming using CUDA

Our GPU-based algorithm was motivated by the fact that it fits very well in the parallel computing model. The voxelization of the axis-aligned bounding box, that encloses the molecule, allows us to speed up the program by assigning a thread to each voxel. Therefore, before proceeding any further, let us first see the relevant details related to GPU programming when using NVIDIA CUDA technology.

3.1.1 CUDA Architecture

CUDA (Compute Unified Device Architecture) API gives data-intensive applications access to the high processing power of NVIDIA graphics processing units (GPUs), unleashing entirely new capabilities through a revolutionary computing architecture. The graphics card used in our work was an NVIDIA GeForce GTX 280. This graphics card possesses 10 thread processing clusters (TPCs), where each TPC consists of 3 streaming multiprocessors (SMs), with each SM having 8 stream processors, which yields 240 cores in total. To control and distribute the massive computing work done by several threads, loaded by the 240 cores, an automatic thread scheduler within the graphics card is used, to guarantee a nearly full utilization of the GPU [8].

3.1.2 CUDA Programming Model

When we are programming using CUDA , more care is needed to make sure that a program is going to be executed correctly. CUDA API extends the C/C++, allowing to program directly the graphics card, in an intuitive way somehow. A CPU-specific function runs on

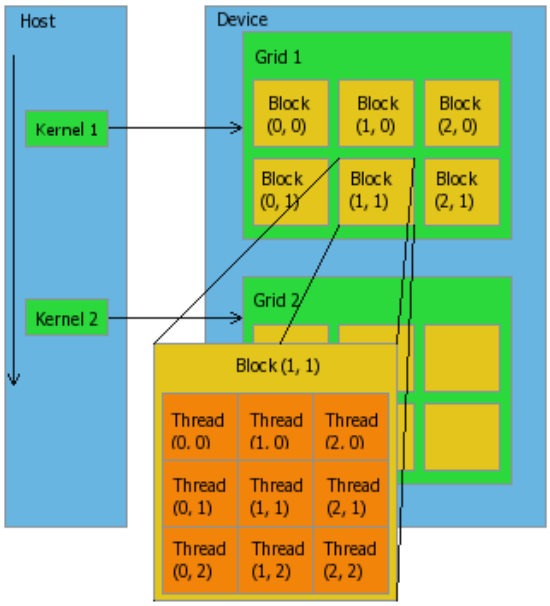

Figure 3.1: GPU programming model (based on an image from [6]).

the CPU (host), while a GPU-specific function is executed on the GPU (device) as show in Figure 3.1.

There are three types of GPU-specific qualifiers for functions and variables. The first, named `__global__`, acts as prefix to identify which functions are GPU kernels (i.e. GPU functions that are invoked from CPU-specific code). The second, named `__device__`, qualifies a GPU function that can be only called from a GPU kernel. Finally, the third qualifier is the keyword `__shared__`, that acts as a prefix to a variable allocated in the shared memory of streaming multiprocessors.

GPU-specific functions are prohibited to be recursive [6, 43], since the CUDA memory management model does not allow global memory allocations from kernels. There are other three CUDA functions to allocate on and free memory from the GPU side, and to exchange data between CPU and GPU. The `cudaMalloc` function is used to allocate memory on graphics card, while the `cudaFree` function releases memory space from graphics card. The `cudaMemcpy` function serves to exchange data between the CPU side to GPU side [6, 43]. Note that the GPU runs kernels. A kernel is a GPU-specific function that launches threads, hierarchically organized into grid–blocks–threads, as illustrated in Fig. 3.1.

The compilation of a CUDA program is done in three stages. The first extracts the CPU code from the CUDA file into an intermediate file, which is then passed to the system

standard C compiler. Next, the GPU code, is converted to a PTX file (i.e., a kind of CUDA specific assembly language). The third stage translates this PTX file into GPU-specific commands, and encapsulates them in an executable file.

3.2 Mathematical Formulations of Molecular Surfaces

There are two major mathematical formulations for surfaces, namely: parametric surfaces and implicit surfaces. A *parametric surface* is defined by a function f with domain $\mathbb{R}^2$ and range $\mathbb{R}^3$. The parametric surface is just the image of this function and is generated by varying the parameters u and v in the domain $D \subset \mathbb{R}^2$.

$$f(u,v) = (X(u,v), Y(u,v), Z(u,v))$$

where $X(u,v)$, $Y(u,v)$ and $Z(u,v)$ are the component functions.

An *implicit surface* is defined by a function F with domain $\mathbb{R}^3$ and range $\mathbb{R}$. The implicit surface is given by all the points in $\mathbb{R}^3$ that satisfy the function $F(x,y,z) = T$, where T is the isovalue. That is, an implicit surface corresponds to the level set defined by the constant T. Thus, $F > 0$ for points outside the surface, and $F < 0$ for points inside the surface, and $F = T$ for points of the surface.

Implicit surfaces can take many forms. However, in this thesis we are only interested in the ones that are capable of representing of the electrical intensity field F of the molecule from summing up the local intensity contributions f_i of its N atoms, as follows:

$$F(x,y,z) = \sum_{i=0}^{N} f_i(x,y,z) \tag{3.1}$$

The local contribution of each atom is given by a function that monotonically decreases with the distance from its center. That is, the electrical field of each atom in the molecule is formulated as a distance function, *i.e.* an implicit scalar function. These atomic local functions work as blending functions of the resulting molecular surface, which will be hopefully smooth. The smoothness of the molecular surfaces depends on the smoothness of the blending functions. Thus, the blending functions must be differentiable. A surface with this characteristics is referred to as *convolution surface* [17].

In the literature, we find several blending functions that monotonically decreases with the distance to a center point. Examples are:

- Blinn function.
- Soft object function.
- Inverse squared function.

3.2.1 Blinn Function

The Blinn function is a particular Gaussian given by:

$$f_i(x,y,z) = ae^{-br^2} \tag{3.2}$$

where b is the standard deviation of Gaussian curve, a is the height of Gaussian Curve, and $r = \sqrt{(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2}$ is the distance to the atom center (Figure 3.2).

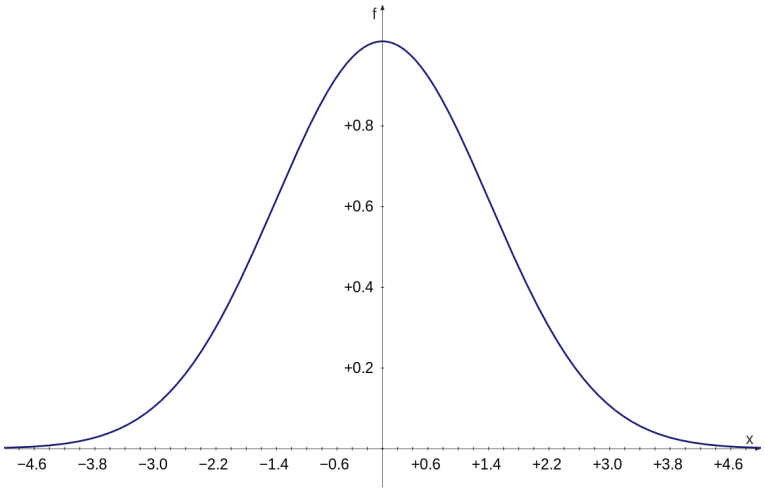

Figure 3.2: Blinn function.

As a Gaussian function, the Blinn function is symmetric, and quickly falls off, being zero at infinity. Recall that the Blinn function characterizes the electron density around an atom. For N atoms, the density function at a given surface point is obtained by summing up the contributions f_i of all atoms:

$$F(x, y, z) = \sum_{i=0}^{N} a_i e^{-b_i r^2} = T \tag{3.3}$$

3.2.2 Soft Object Function

The soft object function is similar to the Gaussian function. But, unlike the Gaussian, it is a compactly-supported function, that is, it vanishes at a finite distance from the origin (i.e atom center), as illustrated in Figure 3.3.

The soft object function was proposed by Wyvill et al. [55], and is given by:

$$f_i(x, y, z) = \begin{cases} a(1 - \frac{4r^6}{9b^6} + \frac{17r^4}{9b^4} - \frac{22r^2}{9b^2}) & r \leq b \\ 0 & r > b \end{cases} \tag{3.4}$$

where b is the standard deviation of the curve, a is the height of the curve, and $r = \sqrt{(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2}$ is the distance to the center of atom i. It is a simplification of the Gaussian function because it is obtained by truncating the Taylor expansion series of the exponential function. The final result of this simplification is a less time-consuming polynomial function than the Blinn function.

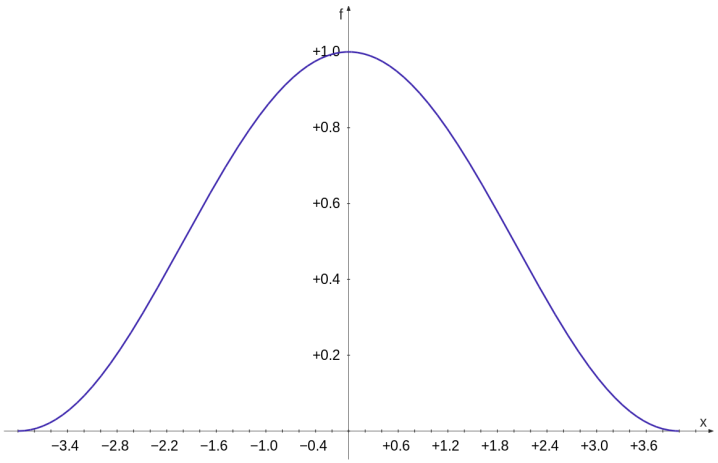

Figure 3.3: Soft object function.

3.2.3 Inverse Squared Function

An even cheaper blending function is the inverse squared function (3.4), which is as follows:

$$f_i(x, y, z) = \frac{C}{r_i} \tag{3.5}$$

where C stands for the smoothness or blobiness parameter, and $r_i = (x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2$ is the squared distance from the center (x_i, y_i, z_i) of the atom i to a generic point (x, y, z). Typically, the parameter C takes on a value in the interval $]0, 1]$.

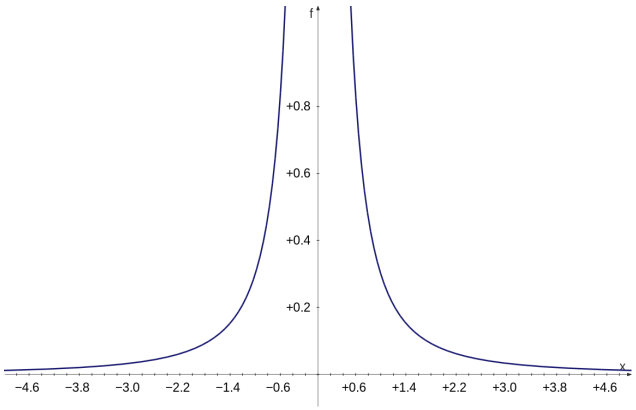

Figure 3.4: Graph of Inverse Squared Function.

3.3 Triangulation of Connolly Surfaces

In the previous section we have addressed the mathematics of molecular surfaces used to represent the electrical field intensity of a molecule as an isosurface. This formulation allows for rendering molecular surfaces using triangulation algorithms.

Amongst all the available triangulation algorithms used to visualize molecular surfaces, the marching cubes (MC) algorithm is possibly the most adequate for GPU processing. This is justified by the fact that MCs lead to an uniform space partition of the bounding box into voxels, which is ideal for the design of a possible parallel implementation on GPU.

3.3.1 Marching Cubes: Overview

This algorithm starts by dividing, the space that contains the molecule into an uniform grid of voxels of appropriate size. The next step is the calculation of the electrical field intensity F (Equation 3.1) at all voxel vertices of the grid. Doing this involves very time-consuming computation, due to the fact that organic molecules have long chains of atoms, and the spatial distribution of these atoms often requires very large grids, even for molecules with a small number of atoms.

To deal with this problem, our algorithm calculates the intensity in a per atom charge fashion, instead of calculating it at each grid vertex directly. Since the influence of the field can be neglected beyond some appropriate distance, this optimization can be used in general for intensity calculations. Next, we calculate the position of the voxel for each atom, and then we define a sub-grid centered on such a atom/voxel. The intensity contribution of this atom adds to the current intensity of each sub-grid vertex.

Terminated the computation of the intensity F for all grid vertices, we have to determine the 8-bit flag for each voxel. This flag caracterises the surface inside each voxel. The flag bits form an index to a lookup table to determine which edges of each voxel intersect the isosurface. If $F \geq T$ at a grid vertex, the corresponding flag bit is set to 1; otherwise it is set to 0. After, setting the flag bits for the each voxel, the algorithm uses the lookup table to determine which of the 256 cases of surface fits inside the voxel. When this is done, the algorithm goes a step forward to interpolate the surface vertices, for each cube along the appropriate edges, by using the lookup tables. However, simply interpolating the vertices along edges for each cube will cause the generation of the same vertex information four times, as each edge is shared by four cubes. In order to avoid duplication of mesh vertices, we use a hash key generated from the 6 coordinates of the vertices bounding such an edge (intersecting the surface) to check whether or not such a surface vertex has been already determined. If no vertex corresponding to the key exists, a new vertex is interpolated along the edge. Note that the re-computation of surface vertices could not be avoided in the GPU-based implementation because we do not known in advance whether the threads that process neighboring voxels have finished or not. This is a difficult synchronization problem to fix in multi-threaded parallel computing on GPU.

After computing the surface vertices for all voxels, we are ready to generate the triangles inside each voxel, which together form the the final mesh that approximates the molecular surface. It is clear that rendering this surface mesh requires the calculation of the triangle

normals. Next, we describe the implementation for the Connolly Surface, and for the Van Der Waals surface, both done either on the CPU and on the GPU.

3.3.2 CPU implementation

A CPU-based marching cubes algorithm for Connolly surfaces has been implemented using C++. All MC computations have been performed by the CPU, and the algorithm can be described as follows:

- *Bounding Box Computation*. First we determine the axis-aligned bounding box that encloses a given molecule. This is done while reading the center of each atom, from a PDB file, into an 1-dimensional array, of size N, where N is the number of atoms of the molecule. The computation of the bounding box means computing the locations of the opposite vertices to the diagonal of the bounding box.
- *Voxel Decomposition*. Next, the determination of the number $N_V = I \times J \times K \times$ of cubic voxels that fit the bounding box is performed, with a predefined size, where I, J, and K stand for the number of voxels along x-, y-, and z-axis, respectively.
- *Computation of Electric Field Intensities at Voxel Vertices*. At this stage of the algorithm, the intensity F of the electric field at each voxel vertex is calculated. To do this, the initial value at each vertex must be first initialized to 0. Then, this vertex intensity is iteratively updated with the contribution of each atom.

Algorithm 1 Electrical Field Intensity

Input:
N: number of atoms
C: blobiness constant
Output:
intensities values of $F[i][j][k]$

1: **for** $n = 1$ to N **do**
2: Calculate the voxel position (i, j, k) where the atoms lies in.
3: ▷ computation of sub-grid with $20 \times 20 \times 20$ voxels
4: **if** ($i >$ 20) **then**
5: $i_{min} = i - 20$
6: **else**
7: $i_{min} = 0$
8: **end if**
9: ...
10: **if** ($k >$ 20) **then**
11: $k_{max} = k + 20$
12: **else**
13: $k_{max} = 0$
14: **end if**

Algorithm 2 Electrical Field Intensity (cont.)

15: ▷ calculate the intensity
16: **for** $i = i_{min}$ to i_{max} **do**
17: **for** $j = j_{min}$ to j_{max} **do**
18: **for** $k = k_{min}$ to k_{max} **do**
19: $r = (x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2$
20: $F[i][j][k] = C/r$
21: **end for**
22: **end for**
23: **end for**
24: **end for**

As shown in Algorithm 2, we first calculate the position i, j, k of the cube that contains each atom centered at x_i, y_i, z_i (step 2). Then, we define a local sub-grid centered at each atom voxel with $20 \times 20 \times 20$ voxels (step 3 to 14). Thus, we are assuming that the local blending function of any atom outside its surrounding sub-grid takes on the value 0. Them, the intensity contribution f_i of the atom i at each sub-grid vertex is added to the current intensity value at such a vertex (steps 15 to 23).

- *Computation of MC Configurations for Voxels*. After computing the intensities f_i of grid vertices, the next step is to determine the 8-bit flag that corresponds to 1 out of 256 MC surface patterns inside each voxel.

Algorithm 3 8-Bit Flag

Input:
N_{VX}: number of bounding box voxels in x direction
N_{VY}: number of bounding box voxels in y direction
N_{VZ}: number of bounding box voxels in z direction
T: threshold isovalue
Output:
Voxel flag: $flag[i][j][k]$

1: **for** $i = 0$ to N_{VX} **do**
2: **for** $j = 0$ to N_{VY} **do**
3: **for** $k = 0$ to N_{VZ} **do**
4: **if** $(F[i][j+1][k+1] < T)$ **then**
5: $flag[i][j][k] = flag[i][j][k] \mid 1;$
6: **end if**
7: **if** $(F[i+1][j+1][k+1] < T)$ **then**
8: $flag[i][j][k] = flag[i][j][k] \mid 2;$
9: **end if**
10: **if** $(F[i+1][j+1][k] < T)$ **then**
11: $flag[i][j][k] = flag[i][j][k] \mid 4;$
12: **end if**

Algorithm 4 8-Bit Flag (cont.)

```
13:             if (F[i][j + 1][k] < T) then
14:                 flag[i][j][k] = flag[i][j][k]  |  8;
15:             end if
16:             if (F[i][j][k + 1] < T) then
17:                 flag[i][j][k] = flag[i][j][k]  |  16;
18:             end if
19:             if (F[i + 1][j][k + 1] < T) then
20:                 flag[i][j][k] = flag[i][j][k]  |  32;
21:             end if
22:             if (F[i + 1][j][k] < T) then
23:                 flag[i][j][k] = flag[i][j][k]  |  64;
24:             end if
25:             if (F[i][j][k] < T) then
26:                 flag[i][j][k] = flag[i][j][k]  |  128;
27:             end if
28:         end for
29:     end for
30: end for
```

Algorithm 3 iterates on the grid of voxels to determine the 8-bit flag that characterizes the shape of the surface within each voxel. This flag is initialized to 0 (decimal), i.e. 00000000 (binary). This flag results from the bitwise OR of eight masks, each per vertex of the current voxel. For example, if the intensity function at the vertex 3 evaluates to a value less than the pre-defined isovalue T (threshold), its mask is then 8 in decimal or 00001000 in binary (step 10 to 12); otherwise, it is 00000000. This 8-bit flag works as an index to the lookup table, for determining which edges of the voxel intersect with molecular isosurface.

- *Surface Triangulation per Voxel.* The previous 8-bit flag works as an index to the $Etable$ lookup table. This table contains 256 configurations in terms of voxel edges that intersect the surface or not. Each configuration is nothing more than a 12-bit mask, a bit per edge. We use each one of these configurations to retrieve the triangles to be generated inside voxel from another lookup table, named $Ttable$. For example, the following $Ttable$ entry $\{1, 8, 3, 9, 8, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1\}$ indicates that we have two triangles inside the voxel. The first triangle is defined by the surface vertices in the edges 1, 8, and 3, while the second is given by those in 9, 8, and 1. But, this requires a prior computation of the surface vertices, i.e. the triangulation vertices that result from the intersection between the surface and the voxel edges. The computation of each triangulation vertex is performed by linear interpolation along one each transverse voxel edge. However, this interpolation needs to first check whether such a vertex has already been calculated or not. This is done using an hash table, where all already determined vertices are saved. Thus, a triangulation vertex is only performed if each vertex does not exist in the hash table.

Algorithm 5 Surface triangulation per voxel

Input:
N_{VX}: number of bounding box voxels in x direction
N_{VY}: number of bounding box voxels in y direction
N_{VZ}: number of bounding box voxels in z direction
$Etable$: edge lookup table with 256 cases of marching cubes
$Ttable$: triangle lookup table
V:array of interpolated vertices
q: counter variable that allow to know the next position fill in the array
Output:
$MESH$: linear array of vertices for the generated triangles

1: **for** $i = 0$ to N_{VX} **do**
2: **for** $j = 0$ to N_{VY} **do**
3: **for** $k = 0$ to N_{VZ} **do**
4: **if** ($Etable[flag[i][j][k]]\&1$) **then**
5: $V[0] = VertexInterpolation(i, j+1, k+1, i+1, j+1, k+1)$;
6: **end if**
7: **if** ($Etable[flag[i][j][k]]\&2$) **then**
8: $V[1] = VertexInterpolation(i+1, j+1, k+1, i+1, j+1, k)$;
9: **end if**
10: **if** ($CTable[flag[i][j][k]]\&4$) **then**
11: $V[2] = VertexInterpolation(i, j+1, k, i+1, j+1, k)$;
12: **end if**
13: **if** ($CTable[flag[i][j][k]]\&8$) **then**
14: $V[3] = VertexInterpolation(i, j+1, k+1, i, j+1, k)$;
15: **end if**
16: **if** ($CTable[flag[i][j][k]]\&16$) **then**
17: $V[4] = VertexInterpolation(i, j, k+1, i+1, j, k+1)$;
18: **end if**
19: **if** ($CTable[flag[i][j][k]]\&32$) **then**
20: $V[5] = VertexInterpolation(i+1, j, k+1, i+1, j, k)$;
21: **end if**
22: **if** ($CTable[flag[i][j][k]]\&64$) **then**
23: $V[6] = VertexInterpolation(i, j, k, i+1, j, k)$;
24: **end if**
25: **if** ($CTable[flag[i][j][k]]\&128$) **then**
26: $V[7] = VertexInterpolation(i, j, k+1, i, j, k))$;
27: **end if**
28: **if** ($CTable[flag[i][j][k]]\&256$) **then**
29: $V[8] = VertexInterpolation(i, j+1, k+1, i, j, k+1)$;
30: **end if**

Algorithm 6 Surface triangulation per voxel (cont.)

```
31:             if (CTable[flag[i][j][k]]&512) then
32:                 V[9] = VertexInterpolation(i+1, j+1, k+1, i+1, j, k+1);
33:             end if
34:             if (CTable[flag[i][j][k]]&1024) then
35:                 V[10] = VertexInterpolation(i+1, j+1, k, i+1, j, k);
36:             end if
37:             if (CTable[flag[i][j][k]]&2048) then
38:                 V[11] = VertexInterpolation(i, j+1, k, i, j, k);
39:             end if
40:             for (n=0 to Ttable[flag[i][j][k]]][n]!=-1) do
41:                 MESH[q] = V[Ttable[flag[i][j][k]][n]];
42:                 MESH[q+1] = V[Ttable[flag[i][j][k]][n+1]];
43:                 MESH[q+2] = V[Ttable[flag[i][j][k]][n+2]];
44:             end for
45:         end for
46:     end for
47: end for
```

After finding the triangulation vertices associated to a voxel, we are ready to generate its triangles. These triangles are saved in an array, as needed for rendering.

- *Computation of Surface Normals at the Vertices*. However, to render the surface, we first need to compute the surface normal at each surface vertex stored in a vector of vertices. The normal vector at each surface vertex is given by the gradient vector as follows:

$$\nabla F = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z}\right) \tag{3.6}$$

where

$$\frac{\partial F}{\partial x} = \sum_{i=1}^{N} \frac{-2\,C\,(x - x_i)}{[(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2]^2}$$

$$\frac{\partial F}{\partial y} = \sum_{i=1}^{N} \frac{-2\,C\,(y - y_i)}{[(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2]^2}$$

$$\frac{\partial F}{\partial z} = \sum_{i=1}^{N} \frac{-2\,C\,(z - z_i)}{[(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2]^2}$$

These normals are stored in a separate array. Recall that F is the implicit function that describes molecular isosurface given by Equation (3.5).

- *Rendering the Molecular Surface*. This final step iterates on both array of triangles and array of normals to render the the surface mesh. Our implementation uses OpenGL with Gouraud shading.

3.3.3 GPU Implementation

CUDA API was used to implement the prior CPU-based version of the Marching Cubes (MC) algorithm for molecular surfaces. All MC computations are carried out on the GPU. The algorithm can be described as follows:

- *Bounding Box Computation and Voxel Decomposition (CPU side).* These two steps are identical to the ones of the sequential version.

- *Allocation and Copying of Data Structures to GPU*. This step allocates GPU memory for two 1-dimensional arrays, to where we copy the array of N atomic centers, and the array of N_V integers, one integer per voxel. These two operations are carried out by calling the cudaMallloc and cudaMemcpy functions from the CPU side:

```
CUDA_SAFE_CALL(cudaMalloc((void**)&d_nvvoxel_scan,(Voxel1)*sizeof(int)));
CUDA_SAFE_CALL(cudaMemset(d_nvvoxel_scan,0,(Voxel1)*sizeof(int)));
```

  These two functions are also used to allocate and copy the lookup tables of the MC algorithm, as well as the array required for all the voxel vertices, into the GPU memory. The previous mentioned array of N_V integers serves to store the number of mesh vertices of the molecular surface (i.e. patch inside each voxel), as needed in a later step.

- *Computation of Electric Field Intensities at Voxel Vertices* (1st GPU kernel). This computation has exactly the same logic as the one of Algorithm 2. But now we have to use the qualifier `__global__` for this method, as it is a kernel. The kernel body is exactly the same as the body of the first **for** cycle of Algorithm 2. This means that the computation of intensities around each atom is carried out by the GPU thread in parallel. Obviously, this requires the creation of a grid of blocks of threads, as in Figure 3.1.

- *Computation of MC Configurations for Voxels* (2nd GPU kernel). After computing the intensities f_I, on vertices of each voxel, we determine the 8-bit flag that corresponds to one out of 256 MC surface patterns that can be inside each voxel. This kernel is also very similar to the corresponding CPU algorithm 3, with the difference that it processes simultaneously a certain number of voxels. Next it is check the number of vertices for each voxel. This is done through the consulting of a lookup table $tabela$, and using the 8-bit flag of each voxel previously calculated, we obtain the number of vertices to each voxel, that approximates the molecular surface. This lookup table gives the number of the vertices, for each one of the 256 configurations. These numbers will be mapped into the array $voxel$, which gives for a certain voxel the number of vertices used to triangulate the molecular surface inside that voxel (Algorithm 7).

Algorithm 7 Checks each voxel to know the number of vertices

Input:
Pos: coordinates of the voxel that is being analyzed
$tabela$: lookup table with the number of the vertices, for each one of the 256 configurations
$cube$: flag bit corresponding to the voxel
$tex1Dfetch$: pre defined function that allow to consult a lookup table
Output:
$voxel$: Number of vertices in each voxel

1: $nvertices1 = tex1Dfetch(tabela, cube)$
2: **if** $n\_vertices > 0$ **then**
3: $\quad voxel[Pos] = nvertices1;$
4: **end if**

- *Allocation of VBO Array for Posterior Rendering of Surface Mesh* (3rd GPU kernel). By performing a Parallel Prefix Sum (scan) [34], of the number of mesh vertices stored in the array $voxel$, we determine the total number of sampled vertices which will be used to triangulate the molecular surface.

```
cudppScan(scanplan, nvvoxel_scan, nvvoxel, Voxel1);
```

  Recall that this triangulation will be done locally, inside each voxel, as usual in the MC algorithm. This Parallel Prefix Sum operation, is carried out by calling the function `cudppScan`, which is another GPU kernel. The number of vertices, given in the output of `cudppScan`, is then used to allocate a VBO (Vertex Buffer Objects) array for the mesh vertices, from which we can later generate triangles, to render the mesh, that approximates the molecular surface.

```
// create a buffer object
glGenBuffers(1, vbo);
glBindBuffer(GL_ARRAY_BUFFER, *vbo);

// initialize a buffer object
glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

- *Setting up a mapping array between, an array of V_N voxel numbers, and an VBO array (4th GPU kernel).* In this kernel an auxiliary array of N elements is created. This array is used to map between the array of V_n voxel numbers, and the corresponding VBO array. Thus, for each voxel, the mapping array vac stores the index of the VBO array element, which will later be used to store the surface vertex associated with each voxel. This allows an association, between the occupied voxels (e.g. voxels transverse to the molecular surface), and the corresponding position in the VBO array (Algorithm 8).

Algorithm 8 Setting up a Mapping Array

Input:
Pos: coordinates of the voxel that is being analyzed
Output:
Mapping array vac

1: **if** $(voxel[pos] > 0)$ **then**
2: $\quad vac[nvvoxelscan1[pos]] = pos;$
3: **end if**

- *Computation of Surface Vertices by Linear Interpolation* (5th GPU kernel). This GPU kernel uses linear interpolation, along the voxel edges, that intersect the molecular surface, in order to sample the surface.

Algorithm 9 Computation of Surface Vertices by Linear Interpolation

Input:
$nvertices1$: number of vertices in each voxel
$tabelacasos$: edge lookup table with 256 cases of marching cubes
$aresta$:stores the position of the interpolated vertice in the $vertlist$
$cube1$: flag bit corresponding to the voxel being analyzed
$mesh\_in$: index to write in the VBO array
vl: array of interpolated vertices
Output:
$MESH1$: array of vertices for the generated triangles

1: $cube = vertice1[Pos];$
2: $nvertices1 = tex1Dfetch(tabela, cube)$
3: **if** $(nvertices1 > 0)$ **then**
4: $\quad vl[0] = VertexInterpolation(x, x+1, y+1, y+1, z+1, z+1, X, Y, Z);$
5: $\quad vl[1] = VertexInterpolation(x+1, x+1, y+1, y+1, z+1, z, X, Y, Z);$
6: $\quad vl[2] = VertexInterpolation(x, x+1, y+1, y+1, z, z, X, Y, Z);$
7: $\quad vl[3] = VertexInterpolation(x, x, y+1, y+1, z+1, z, X, Y, Z);$
8: $\quad vl[4] = VertexInterpolation(x, x+1, y, y, z+1, z+1, X, Y, Z);$
9: $\quad vl[5] = VertexInterpolation(x+1, x+1, y, y, z+1, z, X, Y, Z);$
10: $\quad vl[6] = VertexInterpolation(x, x+1, y, y, z, z, X, Y, Z);$
11: $\quad vl[7] = VertexInterpolation(x, x, y, y, z+1, z, X, Y, Z);$
12: $\quad vl[8] = VertexInterpolation(x, x, y+1, y, z+1, z+1, X, Y, Z);$
13: $\quad vl[9] = VertexInterpolation(x+1, x+1, y+1, y, z+1, z+1, X, Y, Z);$
14: $\quad vl[10] = VertexInterpolation(x+1, x+1, y+1, y, z, z, X, Y, Z);$
15: $\quad vl[11] = VertexInterpolation(x, x, y+1, y, z, z, X, Y, Z);$
16: **end if**
17: **for** $(i = 0$ to $nvertices)$ **do**
18: $\quad aresta = tex1Dfetch(Etable, cube1);$

Algorithm 10 Computation of Surface Vertices by Linear Interpolation (cont.)

19: $v[0].x = vl[aresta].x$;
20: $v[0].y = vl[aresta].y$;
21: $v[0].z = vl[aresta].z$;
22: ...
23: $MESH1[mesh\_in] = make\_float4(v[0].x, v[0].y, v[0].z, 1.0f)$;
24: ...
25: **end for**

To generate the surface, for each voxel, we must first know the number of mesh vertices (step 2), that a certain voxel has, using the lookup table $tabelacasos$. If that number is above 0 we are going to process that voxels (step 3), otherwise we are not. When the number of vertices is above 0, we must find the vertices, where the surface intersects the cube. This calculation is done with linear interpolation, of the twelve edges that are part of a cube (step 4 - 15). When all the twelve results of the linear interpolation are available we must see which are the edges were the surface pass. To obtain this information, for each vertice, we use a pre-defined lookup table $tabelacasos$, from where we obtain the number of the edge, that passed through the surface (step 18). With this number we must go to the vector, that has the results of the interpolation for that voxel, and associate the edge number, to the corresponding position in the interpolation vector (step 19 - 21). The final position of a vertice is stored in the corresponding position, in the buffer object, that is responsible to save all the vertices positions (step 23).

- *Computation of Surface Normals at the Vertices in VBO Array* (6th GPU kernel). Taking into account that the stored vertices, in the VBO array, are points of the molecular surface. Where the normal of a given surface point is given by the equations 3.6, 3.3.2, 3.3.2, 3.3.2.

- *Rendering the Molecular Surface (CPU side).* Taking into consideration that the VBO array in the GPU can be accessed from the CPU side, we have only to display the VBOs in order to render the surface. As previously mentioned Gouraud shading that comes with OpenGL is used.

3.4 Triangulation of van der Waals Surface

The triangulation of the van der Waals surface of a molecule is done in two steps:

- Triangulation of the Connolly surface using the MC algorithm.

- Subdivision-based smoothness procedure using a Newton corrector method.

This algorithm as the advantage of not computing the intersection between the spherical atoms. In the first step, we have reduce the size of the sub-grids around each atom in the

attempt of reducing the total processing time of the algorithm. Unfortunately, the result is a non-smooth molecular surface, as shown in Figure 3.5.

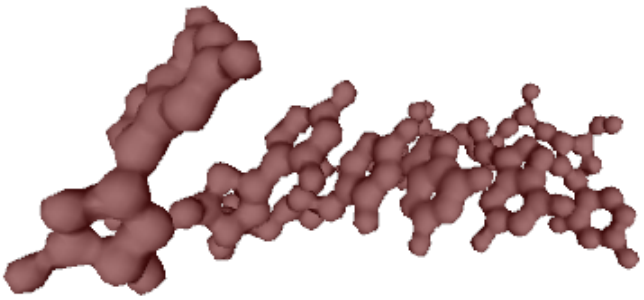

Figure 3.5: Example of a molecule with no smooth.

To resolve this poor shape problem, we have introduced a second step in the algorithm. This step caries out a mid-edge subdivision method. The mid-edge subdivision method divides a triangle into four triangles recursively [48]. However, to use the mid-edge subdivision method it is necessary to guarantee that the new generated points are close to the surface. Doing so, the Newton corrector method can be applied to these new points in order to pull them towards the surface. This convergence procedure is given by the following Newton corrector:

$$p_{k+1} = pk - \frac{f(p_k)}{\bigtriangledown f(p_k)^2} \bigtriangledown f(p_k) \tag{3.7}$$

where p is a point in the 3D space, and $\bigtriangledown f(x)$ defines the gradient vector at any point in space, including the surface itself. The result of this second step for the molecule shown in Figure 3.5 is shown in Figure 3.6.

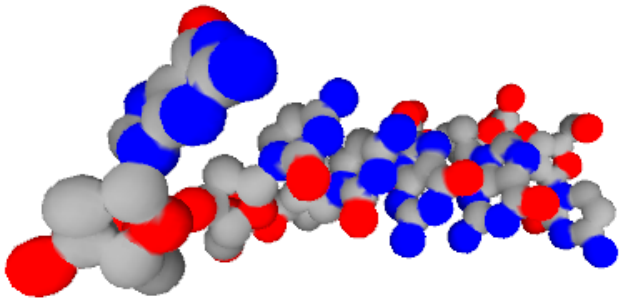

Figure 3.6: Example of a Van Der Waals Surface.

To triangulate a van der Waals surface we must change the Marching Cubes algorithm which was applied to the Connolly surface. The modifications were made in the Algorithm 5 before the creation of the triangles (step 40). In this part of Algorithm 5, we must apply a sub-division of the mesh generated by the MC algorithm to produce a smoother shape for

the van der Waals surface. Each triangle is subdivided into four new triangles recursively, by taking the midpoints of the original triangle edges. These edge midpoints will be the new vertices of the triangulation, to which we then apply the Newton corrector to pull them towards the surface. Note that we have also to compute the normals to triangulation vertices for shading purposes. This calculation is straightforward using the gradient vector.

3.4.1 CPU - Algorithm

The CPU-based Marching Cubes Algorithm, used to represent the van der Waals surface, was implemented using the C++ programming language. The algorithm can be describe as follow:

- *Bounding Box Computation, Voxel Decomposition, Computation of Electric Field Intensities at Voxel Vertices, Computation of MC Configurations for Voxels*. These five steps are identical to the ones of the CPU implementation for Connolly Surface.

- *Computation of Surface Vertices by Linear Interpolation*. In relation to the van der Waals surface CPU-based Algorithm 5, the first part of this method is equal to the one described for the Connolly surface (steps 1-39). The differences between them starts after the interpolation is performed. In this new part of the code new points are calculated, either four or sixteen, depending on the intended level of Mid-edge sub-divisions. When those points are calculated, an approximation of those points to the surface is done, using the Newton Corrector Method. However, to use the Newton corrector method, it must be ensured that the point being analyzed is close enough to the surface. If this point is not close enough to the surface, when performing interpolation, its new values wont converge to the surface. To guarantee this, before applying the Newton method, we have to check whetter or not the points are near the surface. When the point is near the surface, the Newton method is applied, and the identification of the corresponding atom is done. After this, the triangles to each one of the voxels are created, and saved in an array, to be further used at the rendering stage.

- *Computation of Surface Normals at the Vertices*. This step is identical to the one of the CPU-based implementation of the Connolly surface.

- *Computing of the Surface Colors for each atom*. The color to apply to each vertex of each atom is determined by the color associated to such an atom in chemistry.

- *Rendering the Molecular Surface*. This step is identical to the one done of the CPU-based implementation for the Connolly Surface.

3.4.2 GPU Algorithm

The GPU-based implementation of the Marching Cubes algorithm for molecular van der Waals surfaces was carried out using the following algorithm:

- *Bounding Box Computation, Voxel Decomposition ,Allocation and Copying of Data Structures to GPU, Computation of Electric Field Intensities at Voxel Vertices, Computation of MC Configurations for Voxels, Allocation of VBO Array for Posterior Rendering of Surface Mesh, Setting up a Mapping Array Between Array of V_n Voxel Numbers and VBO Array*. These steps are identical to the ones done of the GPU-based implementation for Connolly Surfaces.

- *Computation of Surface Vertices by Linear Interpolation (5th GPU kernel)*. This kernel is very similar to the corresponding CPU-based implementation, used in the computation of surface vertices by linear interpolation, with the difference that a certain number of atoms are processed simultaneously in parallel.

- *Computation of Surface Normals at the Vertices in VBO Array*. This step is identical to the one done of GPU-based implementation for Connolly surface.

- *Computing of the Surface Colors for each Atom (7th GPU kernel)*. This kernel is very similar to the corresponding CPU-based algorithm used to compute the surface colours for each atom, with the difference that a certain number of atoms are processed simultaneously in parallel.

- *Rendering the Molecular Surface*. This step is identical to the one of the GPU-based implementation for the Connolly Surface.

3.5 Final Remarks

In this chapter, we have described a GPU-based triangulation algorithm to render molecular surfaces. First, we have introduced the CUDA technology. Then we have addressed the analytical methods of molecular surfaces used to represent and model a molecule through its behavior of electrical field intensity, as an isosurface. Next, algorithms used to visualize molecular surfaces were described. Finally, we use the Marching Cubes algorithm for rendering a molecular surface and the corresponding CPU and GPU-based implementations.

Chapter 4

Experimental Results

This chapter presents the results of the performance tests done with our CPU- and GPU-based algorithms. Both algorithms were implemented using the C++ STL (Standard Template Library). These tests were carried out using a PC with a Quad core Q9550 at 2,83GHz, 4 GB RAM and a Nvidia GeForce GTX 280. To test the performance of the Connolly surface and Van Der Waals surface, sixteen different .pdb files were used, a file per molecule: 110D, 200D, 1QL1, 4PTI, 1BK2, 2QZF, 2QZD, 2OT5, 1HH0, 1HGV, 1HGZ, 2INS, 1QL2, 1IZH, 1GT0, 1BIJ. These files were obtained from the Protein Data Bank repository.

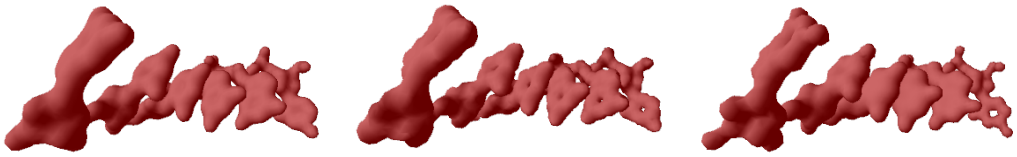

Figure 4.1: Connolly surfaces of the molecule 110d: Blinn function (left), Soft Objects function (center), Inverse Squared function (right)

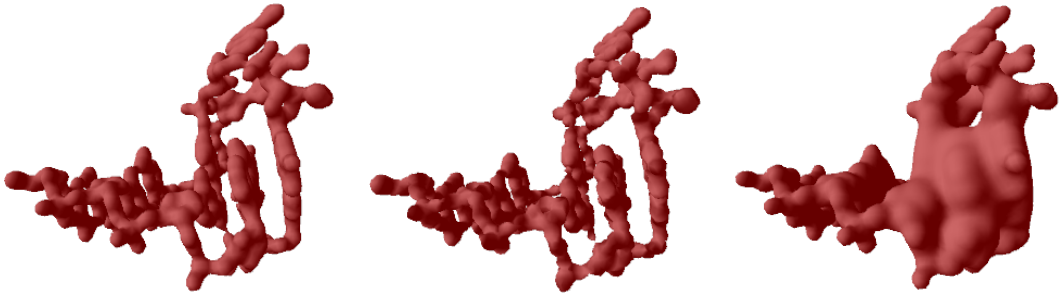

Figure 4.2: Connolly surfaces of the molecule 200: Blinn function (left), Soft Objects function (center), Inverse Squared function (right)

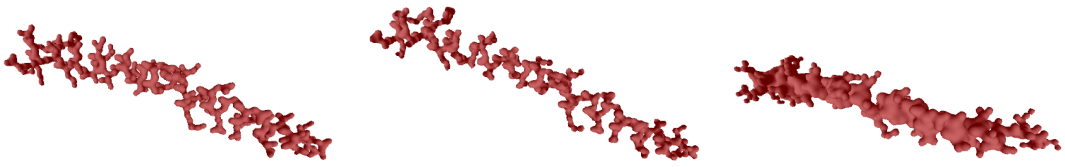

Figure 4.3: Connolly surfaces of the molecule 1ql1: Blinn function (left), Soft Objects function (center), Inverse Squared function (right)

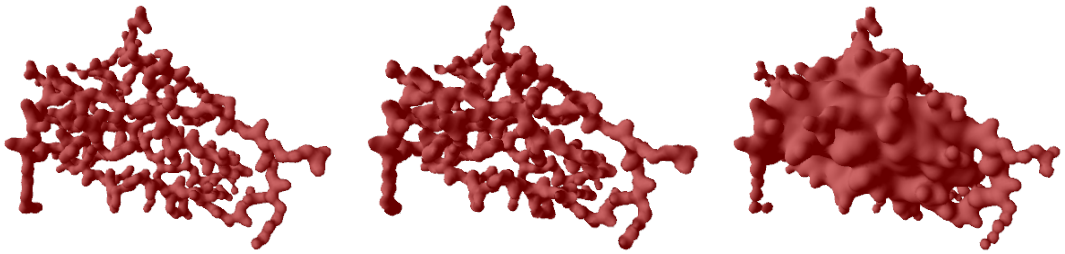

Figure 4.4: Connolly surfaces of the molecule 4pti: Blinn function (left), Soft Objects function (center), Inverse Squared function (right)

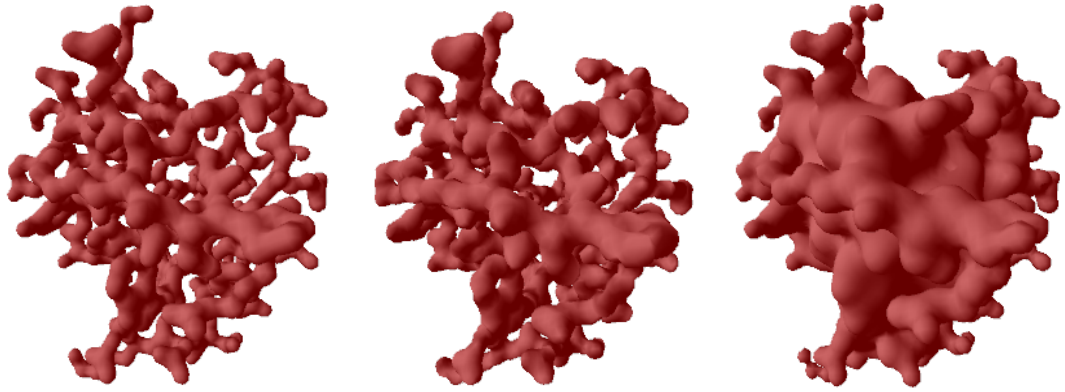

Figure 4.5: Connolly surfaces of the molecule 1bk2: Blinn function (left), Soft Objects function (center), Inverse Squared function (right)

Figure 4.6: Connolly surfaces of the molecule 2qzf: Blinn function (left), Soft Objects function (center), Inverse Squared function (right)

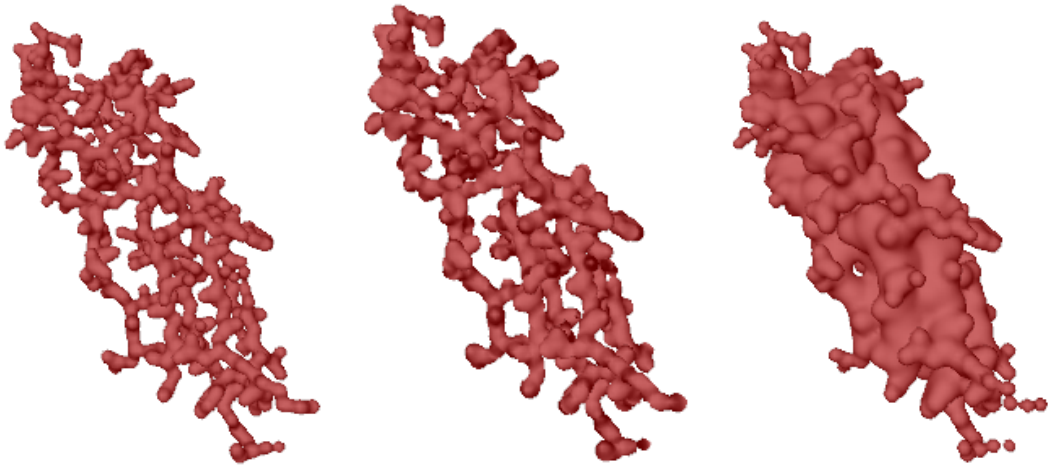

Figure 4.7: Connolly surfaces of the molecule 2qzd: Blinn function (left), Soft Objects function (center), Inverse Squared function (right)

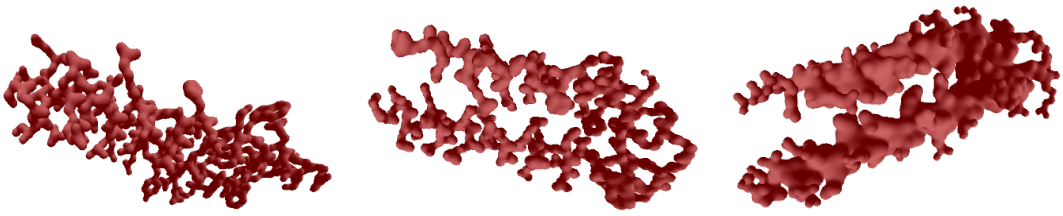

Figure 4.8: Connolly surfaces of the molecule 2ot5: Blinn function (left), Soft Objects function (center), Inverse Squared function (right)

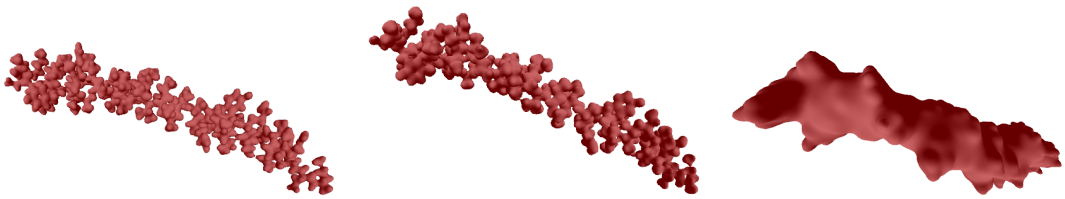

Figure 4.9: Connolly surfaces of the molecule 1hh0: Blinn function (left), Soft Objects function (center), Inverse Squared function (right)

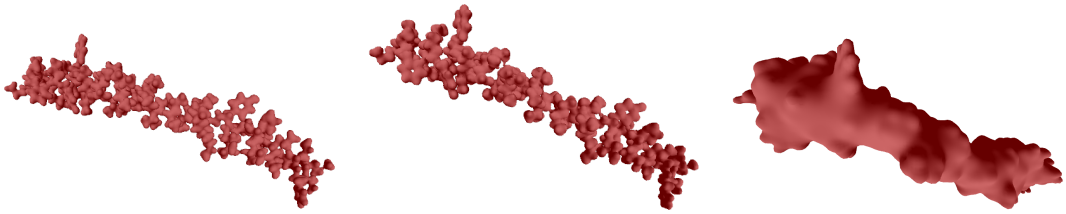

Figure 4.10: Connolly surfaces of the molecule 1hgv: Blinn function (left), Soft Objects function(center), Inverse Squared function (right)

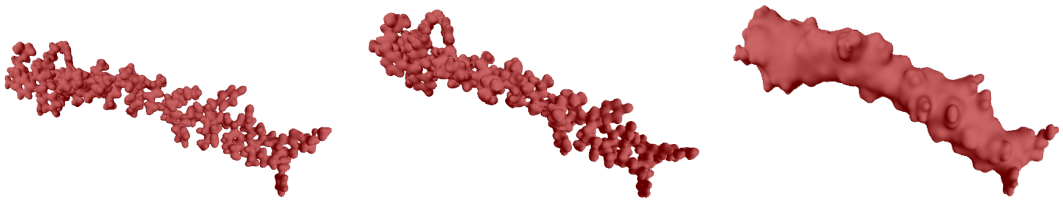

Figure 4.11: Connolly surfaces of the molecule 1hgz: Blinn function (left), Soft Objects function (center), Inverse Squared function (right)

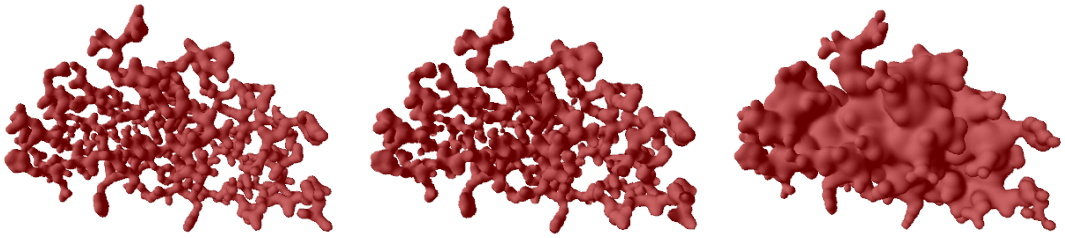

Figure 4.12: Connolly surfaces of the molecule 1hgz: Blinn function (left), Soft Objects function (center), Inverse Squared function (right)

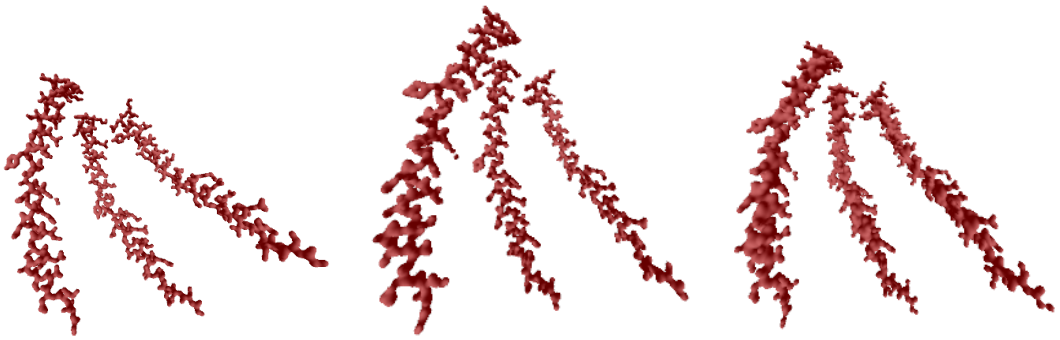

Figure 4.13: Connolly surfaces of the molecule 1ql2: Blinn function (left), Soft Objects function (center), Inverse Squared function (right)

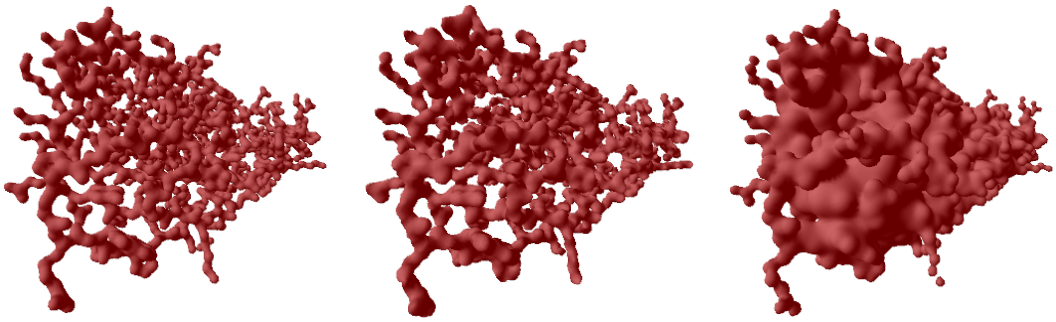

Figure 4.14: Connolly surfaces of the molecule 1izh: Blinn function (left), Soft Objects function (center), Inverse Squared function (right)

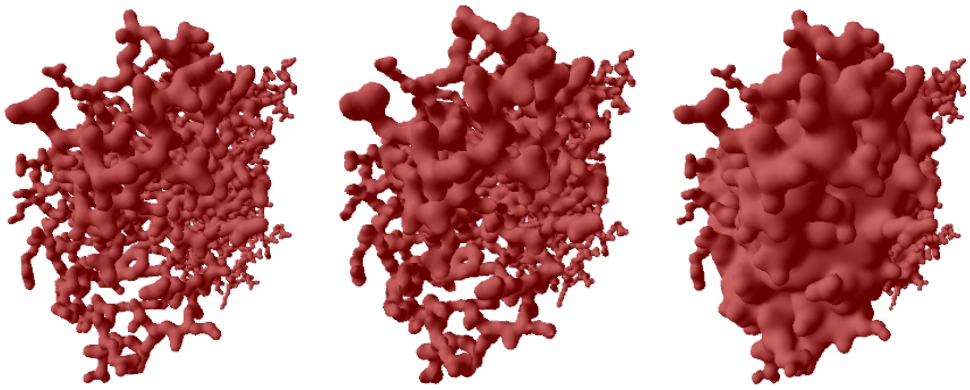

Figure 4.15: Connolly surfaces of the molecule 1gt0: Blinn function (left), Soft Objects function (center), Inverse Squared function (right)

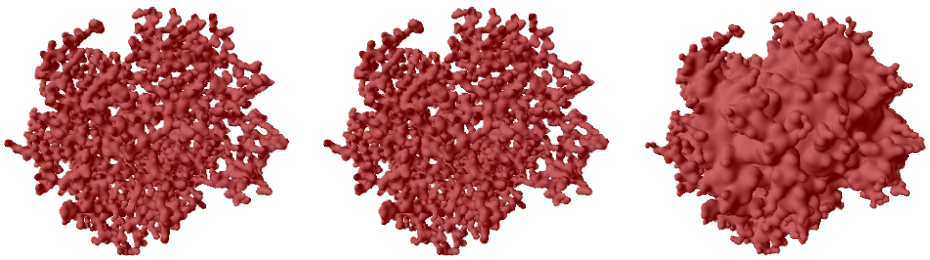

Figure 4.16: Connolly surfaces of the molecule 1bij: Blinn function (left), Soft Objects function (center), Inverse Squared function (right)

4.1 Time Performance of Connolly Surface Triangulation

In this section, we carry out a time performance comparative analysis between the CPU-and GPU-based MC algorithm to render Connolly surfaces, using three different types of blending functions namely: Blinn function, soft object function, and inverse squared function.

4.1.1 Time Performance using CPU

The experimental time results are shown in Table 4.1. Note that these results were obtained without using an hash table that allows to avoid the re-computation of mesh vertices shared by neighbor voxels.

ID	# Atoms	# Triangles	Blinn Function (sec)	# Triangles	Soft Object Function (sec)	# Triangles	Inverse squared Function (sec)	# Voxels
110D	120	14632	0,68	12584	0,45	12852	0,43	346580
200D	259	32130	1,90	27600	1,40	26872	1,20	603520
1QL1	322	43170	2,79	34020	2,14	37072	1,90	1581370
4PTI	381	60910	4,89	48616	3,70	49036	3,00	1043464
1BK2	468	63442	4,89	50514	3,70	48960	3,00	650160
2QZF	479	64292	5,55	51176	4,30	54632	3,60	3536664
2QZD	507	67712	6,10	54572	4,70	59156	4,00	3860800
2OT5	545	73642	6,00	58672	4,80	64168	4,14	1154032
1HH0	691	64816	6,40	66268	5,50	51252	4,46	1632000
1HGV	691	64698	6,56	65880	5,50	51868	4,57	1970752
1HGZ	691	64670	6,30	66014	5,45	49924	4,36	1605760
2INS	781	103940	11,10	83472	9,12	86332	7,30	1316250
1QL2	966	129630	16,10	102288	13,90	111920	11,50	4619920
1IZH	1521	204436	37,40	162554	31,20	176716	25,10	2578680
1GT0	2746	363784	102,00	293806	94,14	281120	67,87	5681590
1BIJ	4387	587078	250,50	470552	240,20	485208	179,10	7229120

Table 4.1: CPU times (in seconds) without hash table.

As easily seen from Table 4.1, there is a considerable difference in time when we use different blending functions. This happens so because each blending function has its own computation cost, which depends on the mathematical operations performed by each blending function. For example, we see that the Blinn function is computationally expensive. This is due to the fact that this function requires the calculation of an exponential for all atoms at the corresponding sub-grid vertices. The formulation of Connolly surfaces using soft object functions, presents better time performance, when compared to Blinn functions, because this function is a simplification from the Blinn function through a Taylor expansion. This means that soft object function is going to have less mathematical operations to be done when compared to the Blinn function.

The last column function (inverse squared), when compared with the two previously analyzed functions, is the fastest to compute Connolly surfaces. In this function all the atoms contribute to the modeling of a surface, and the summation of all these values will not be too much expensive (i.e. time consuming). This is related to the simplicity of the function, which requires only one type of operation to be performed for each atom. Therefore, this function leads to the better performance times when used to process a Connolly surface, when compared with the other two functions. However, when using the original Marching Cubes Algorithm, we have very long times (Table 4.1), to triangulate a molecule with 4387 atoms.

To reduce the times presented in the Table 4.1, we have used a hash table to shorten the calculations that are necessary to triangulate a Connolly surface in CPU. The already calculated vertices are stored in the hash table, avoiding their re-calculation. Thus we will obtain the same final representation of the molecular surface with less processing time

ID	# Atoms	# Triangles	Blob Function (sec)	# Triangles	Soft Object Function (sec)	# Triangles	Inverse squared Function (sec)	# Voxels
110D	120	14632	0,54	12584	0,33	12852	0,33	346580
200D	259	32130	1,30	27600	0,81	26872	0,77	603520
1QL1	322	43170	1,80	34020	1,20	37072	1,14	1581370
4PTI	381	60910	2,89	48616	1,76	49036	1,61	1043464
1BK2	468	63442	2,68	50514	1,72	48960	1,52	650160
2QZF	479	64292	3,25	51176	2,15	54632	1,61	3536664
2QZD	507	67712	3,52	54572	2,34	59156	1,52	3860800
2OT5	545	73642	3,24	58672	2,15	64168	2,00	1154032
1HH0	691	64816	3,80	66268	2,53	51252	2,22	1632000
1HGV	691	64698	3,90	65880	2,61	51868	2,40	1970752
1HGZ	691	64670	3,71	66014	2,50	49924	2,28	1605760
2INS	781	103940	5,40	83472	3,63	86332	3,22	1316250
1QL2	966	129630	7,65	102288	5,46	111920	5,00	4619920
1IZH	1521	204436	14,50	162554	10,47	176716	9,10	2578680
1GT0	2746	363784	34,90	293806	28,45	281120	22,03	5681590
1BIJ	4387	587078	70,80	470552	66,50	485208	52,95	7229120

Table 4.2: CPU times (in seconds) with hash table.

Looking at the results presented in Table 4.2, we can conclude that the use of an hash table in the Marching Cubes Algorithm makes a big difference in the processing time of the algorithm for all three functions. The use of the hash table causes the decay of the required processing time, as it is shown in Table 4.2.

4.1.2 Time Performance using GPU

The results obtained for GPU-based algorithm to visualize the Connolly surface are presented in the Table 4.3.

ID	# Atoms	# Triangles	Blob Function (sec)	# Triangles	Soft Object Function (sec)	# Triangles	Inverse squared Function (sec)	# Voxels
110D	120	15010	0,13	12444	0,13	13024	0,13	352000
200D	259	32384	0,27	27724	0,27	26984	0,26	610176
1QL1	322	43710	0,38	34132	0,37	37380	0,37	1600896
4PTI	381	60910	0,41	48656	0,40	48424	0,40	1053440
1BK2	468	63864	0,36	50724	0,35	48548	0,35	657920
2QZF	479	64600	0,88	51468	0,72	54032	0,72	3562624
2QZD	507	68110	0,94	54748	0,93	58440	0,94	3901824
2OT5	545	73890	0,56	58752	0,53	64164	0,51	1167360
1HH0	691	66728	0,73	67248	0,69	51728	0,67	1652608
1HGV	691	65646	0,89	66836	0,74	51912	0,70	1994752
1HGZ	691	66574	0,68	66768	0,65	50346	0,64	1626112
2INS	781	105064	0,84	83352	0,82	85304	0,82	1326976
1QL2	966	130026	2,10	102540	2,10	112716	2,00	4654208
1IZH	1521	205290	2,40	162596	2,22	173016	2,15	2601472
1GT0	2746	364868	7,52	294540	7,54	276640	6,90	5720832
1BIJ	4387	588108	15,00	471026	14,40	478272	14,20	7267456

Table 4.3: GPU times (in seconds).

Looking at these results we can see an accentuated decay in the processing time of the Connolly surface. This happens as an consequence of the massive computation that the GPU can process in parallel with its multiple arithmetic logic units. These units can process multiple cubes simultaneously. Besides, from Table 4.3, it is clear that the rendering times of molecular surfaces on GPU are similar, regardless of the Blending function,

4.2 Time Performance of van der Waals Surface Triangulation

To visualize a van der Waals surface we have used the inverse squared function since it is the less time-consuming function. When we apply this function to a molecule the obtained surface will look like the one observed Figure 4.17.

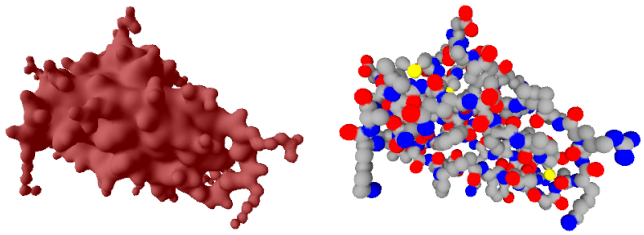

Figure 4.17: Molecule done before apply the Newton Method Corrector (left), Molecule done after apply the Newton Method Corrector (right).

As we can observe from Figure 4.17, we have a Connolly surface, however we want to obtain a van der Waals surface. To do that we must decrease the influence zone of the atoms, and apply the Newton method corrector. This is possible, since when we decrease the influence zone we obtain a surface with a poor shape. However, with a smaller influence zone when applying the Newton corrector method to the surface a visually appealing van der Waals surface is obtained, as illustrated in Figure 4.17.

4.2.1 Time Performance using CPU

The time required to render van der Waals surfaces depends on the number of subdivisions applied to the original MC triangulation. Table 4.4 shows the results of applying to one and two subdivisions to the triangulations of several van der Waals surfaces using the CPU-based implementation.

ID	# Atoms	# Triangles	1 Mid-edge Sub-division (sec)	# Triangles	2 Mid-edge Sub-divisions (sec)	# Voxels
110D	120	33056	0,67	132224	1,69	132526
200D	259	71136	2,33	284544	5,76	280539
1QL1	322	82720	3,18	330880	7,84	880992
4PTI	381	117792	5,83	471168	14,46	558888
1BK2	468	120432	6,10	481728	15,10	309205
2QZF	479	123984	6,60	495936	16,10	2224467
2QZD	507	132624	7,40	530496	18,00	2470608
2OT5	545	139872	8,00	559488	19,80	613824
1HH0	691	205344	14,35	821376	35,80	914085
1HGV	691	206672	14,48	826688	36,10	1148736
1HGZ	691	206368	14,40	825472	36,00	902190
2INS	781	202112	15,60	808448	38,90	737548
1QL2	966	246320	23,40	985280	57,95	3187548
1IZH	1521	391168	55,42	1564672	138,40	1639890
1GT0	2746	724432	179,53	2897728	449,00	4032504

Table 4.4: CPU times of the van der Waals surface.

From the previous results presented in Table 4.4, we conclude that running this algorithm in a CPU is very time-consuming, when processing the van der Waals surface. This happens because of the number of calculations required to obtain the molecular surface. However, this directly dependent from the intended quality for the van der Waals surface. After comparing both results, when using different qualities, we must conclude that the number of triangles is going to influence the computation time, spent to obtain the van der Waals surface. Therefore, to obtain a van der Waals surface with a very good quality, we must wait more time, because of the mathematical calculus that must be performed to obtain the final shape.

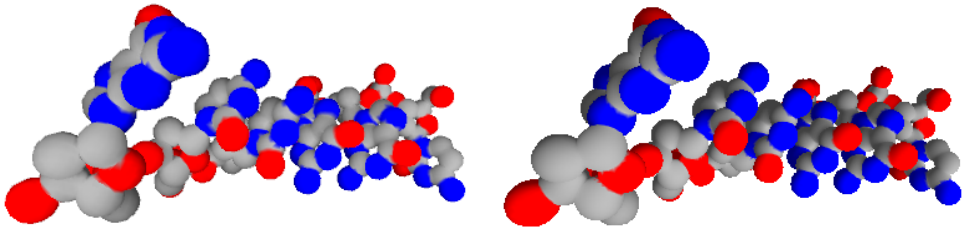

Figure 4.18: van der Waals surface of the molecule 110d: 1 Mid-edge sub-division (left), 2 Mid-edge sub-division (right)

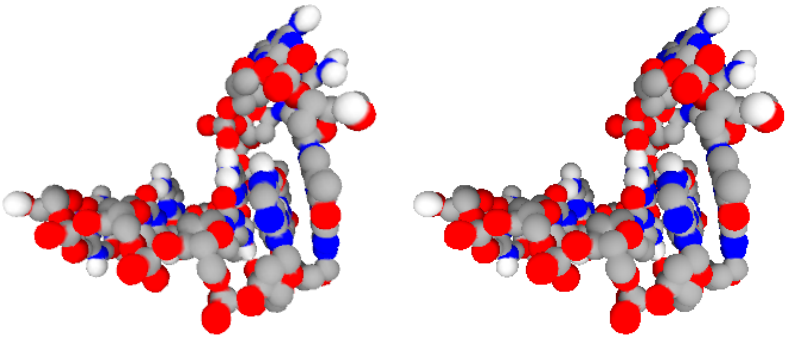

Figure 4.19: van der Waals surface of the molecule 200: 1 Mid-edge sub-division (left), 2 Mid-edge sub-division (right)

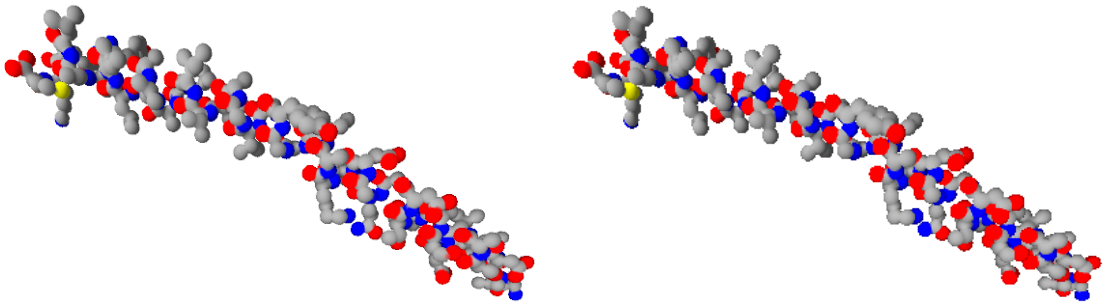

Figure 4.20: van der Waals surface of the molecule 1ql1: 1 Mid-edge sub-division (left), 2 Mid-edge sub-division (right)

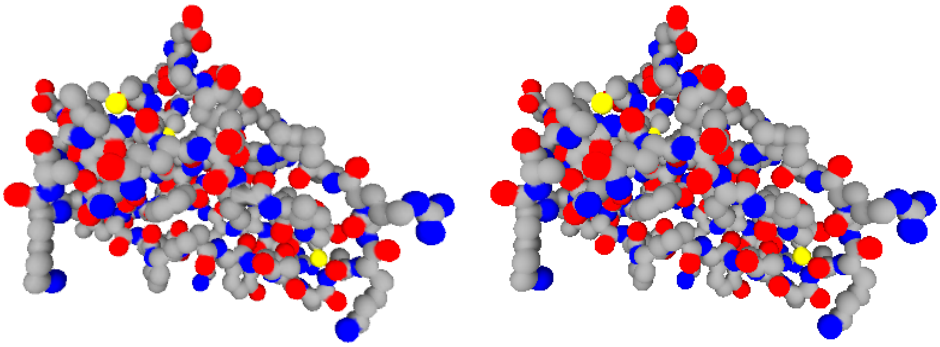

Figure 4.21: van der Waals surface of the molecule 4pti: 1 Mid-edge sub-division (left), 2 Mid-edge sub-division (right)

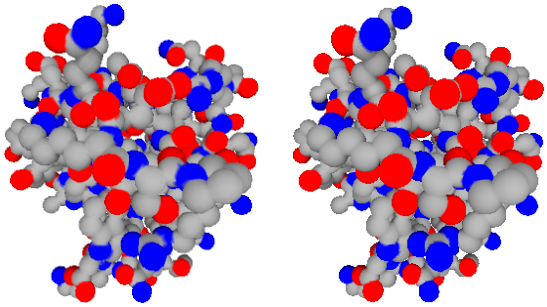

Figure 4.22: van der Waals surface of the molecule 1bk2: 1 Mid-edge sub-division (left), 2 Mid-edge sub-division (right)

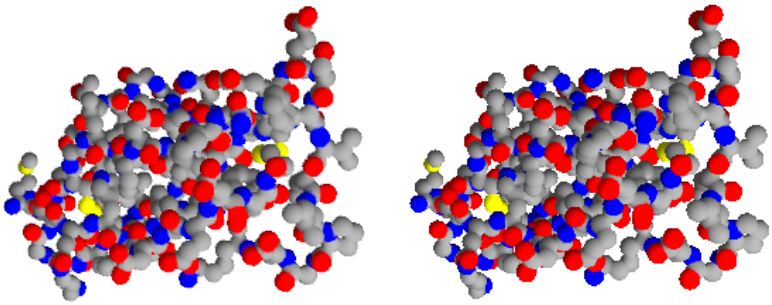

Figure 4.23: van der Waals surface of the molecule 2qzf: 1 Mid-edge sub-division (left), 2 Mid-edge sub-division (right)

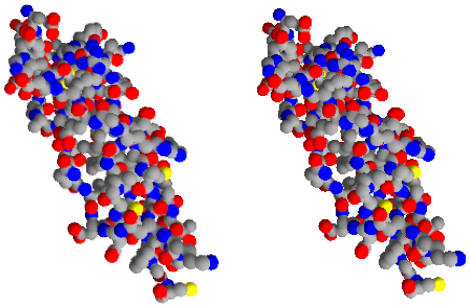

Figure 4.24: van der Waals surface of the molecule 2qzd: 1 Mid-edge sub-division (left), 2 Mid-edge sub-division (right)

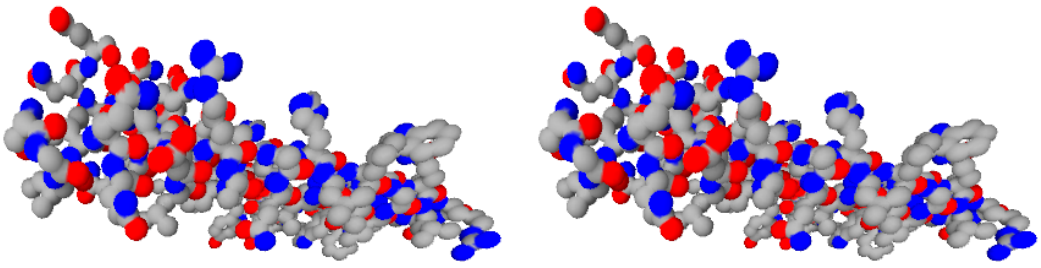

Figure 4.25: van der Waals surface of the molecule 2ot5: 1 Mid-edge sub-division (left), 2 Mid-edge sub-division (right)

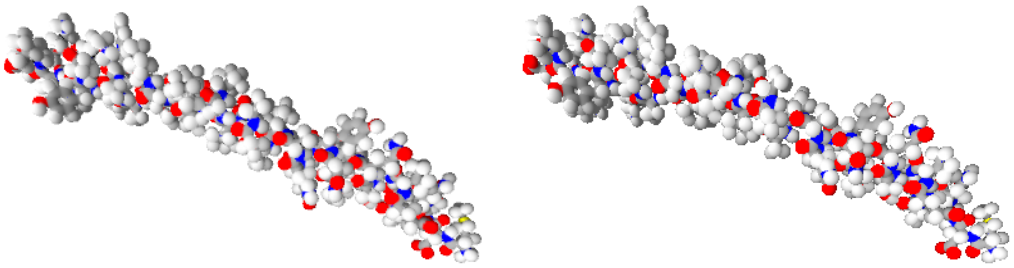

Figure 4.26: van der Waals surface of the molecule 1hh0: 1 Mid-edge sub-division (left), 2 Mid-edge sub-division (right)

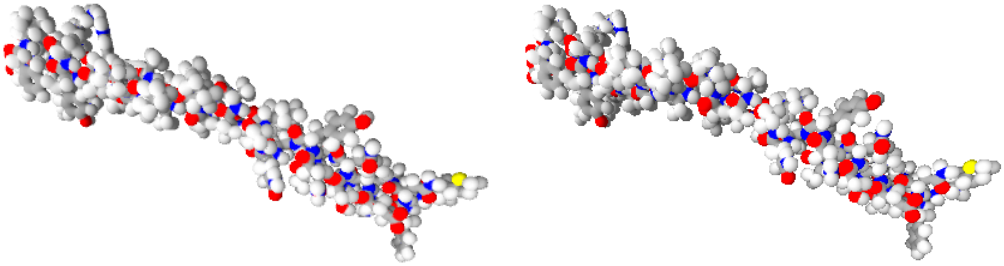

Figure 4.27: van der Waals surface of the molecule 1hgz: 1 Mid-edge sub-division (left), 2 Mid-edge sub-division (right)

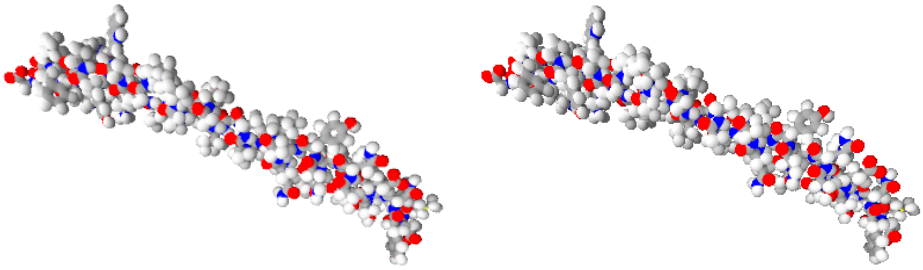

Figure 4.28: van der Waals surface of the molecule 1hgv: 1 Mid-edge sub-division (left), 2 Mid-edge sub-division (right)

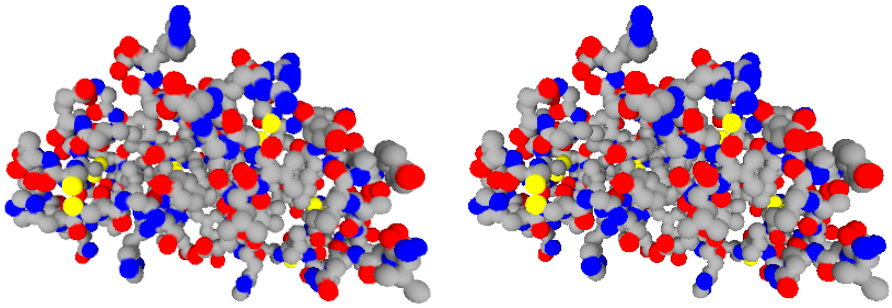

Figure 4.29: van der Waals surface of the molecule 2ins: 1 Mid-edge sub-division (left), 2 Mid-edge sub-division (right)

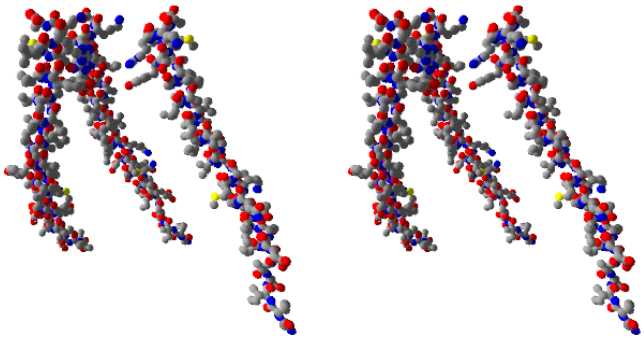

Figure 4.30: van der Waals surface of the molecule 1ql2: 1 Mid-edge sub-division (left), 2 Mid-edge sub-division (right)

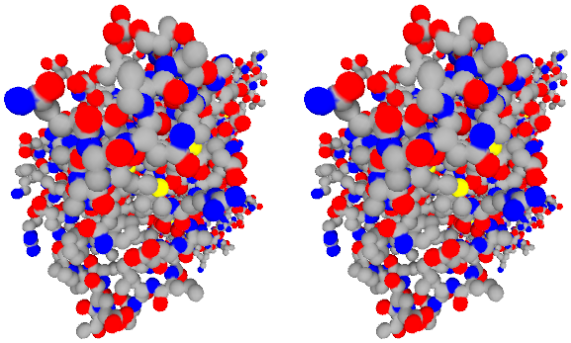

Figure 4.31: van der Waals surface of the molecule 1gt0: 1 Mid-edge sub-division (left), 2 Mid-edge sub-division (right)

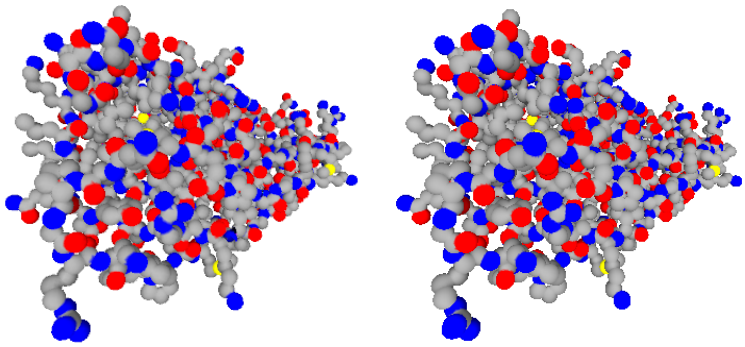

Figure 4.32: van der Waals surface of the molecule 1izh: 1 Mid-edge sub-division (left), 2 Mid-edge sub-division (right)

Analysing the quality of the final surface, obtained for the same molecule, using different numbers of triangles, we can see in Figures 4.7 and 4.8 that we have different qualities. This occurs, because in one version of the algorithm we have very few triangles, and in the other one we have many triangles. However, if we want surfaces with a very good quality, we will have very long processing times in the CPU-based version. Nevertheless, the obtained results obtained from the CPU-based version are still slow, as an consequence of the ineptitude of the CPU to perform high arithmetic calculations.

4.2.2 Time Performance using GPU

Because of the obtained results, we made a GPU-based version of our program. The results are presented in the next table.

ID	# Atoms	# Triangles	1 Mid-edge Sub-division (sec)	# Triangles	2 Mid-edge Sub-divisions (sec)	# Voxels
110D	120	33616	0,08	134528	0,16	135552
200D	259	72512	0,22	290048	0,75	284544
1QL1	322	84848	0,44	339264	1,56	895232
4PTI	381	119024	0,48	474688	1,78	565376
1BK2	468	123744	0,34	494016	1,23	313984
2QZF	479	125728	1,36	502592	4,89	2243456
2QZD	507	134368	1,81	536576	6,79	2503424
2OT5	545	144896	0,61	577984	2,22	622976
1HH0	691	209392	1,03	838144	3,75	929280
1HGV	691	210896	1,13	845120	4,17	1166976
1HGZ	691	209360	0,95	839744	3,53	917248

2INS	781	204048	1,10	817024	4,00	744704
1QL2	966	252096	4,48	1008064	16,36	3215104
1IZH	1521	396128	4,38	1581568	16,56	1657344
1GT0	2746	741856	18,74	2969536	71,98	4064512

Table 4.5: GPU times for the van der Waals surface.

Looking at the Table 4.5, we see a great decrease in the processing time of a van der Waals surface. This decrease results from the GPU capacity to perform many arithmetic mathematical operations in parallel, allowing the processing of several cubes simultaneously. Thus, the GPU architecture is ideal for the computation of problems such as the determination of a van der Waals surface, which will be done in a short time keeping the same quality than the CPU-based version. In relation to the quality of the final image between the two version of the program, the conclusions are equal to the CPU-based version.

4.3 Discussion of Results

Unsurprisingly, the GPU-based implementation of molecular surfaces is faster than the GPU-based one. This happens because the sequential version of the algorithm processes one voxel at a time, while the parallel version processes several voxels at the same time. We also conclude that the Connolly surface is the most efficient function in CPU, since we use the inverse squared function. This shows that the number of mathematical operations needed to process a single blending function influences the time spent to visualize the molecule. However, for the GPU-based implementation this difference it not relevant because the GPU processing is done in parallel. In relation to the van der Waals surfaces, the tests demonstrate that we need a number of mesh subdivisions to get a visually smooth surfaces on screen.

4.4 Final Remarks

In this chapter we have carry out tests to assess the time performance of different functions to visualize Connolly and van der Waals surfaces. This has involved a comparison between a CPU-based implementation and GPU-based implementation for rendering such surfaces.

Chapter 5

Conclusions

Even though geometric modeling is a long established research field that overlaps with computational geometry, computer-aided geometric design, and computer graphics, only recently geometric modelling techniques have started to be applied in molecular modelling. Indeed, a new research area, which we call bio-geometric modeling, emerged from the need to organise and analyse molecular shape information of macromolecules and biological complexes from smaller entities.

As known, biologists and biochemists have long used rigid three-dimensional models to make clear the structure of molecules, which commonly are modelled either as stick-diagrams that emphasise the covalent bonds among atoms, or as space-filling diagrams that describe the space they occupy. In this respect, surfaces play a fundamental role in molecule functions, because chemical-physical actions are driven by their mechanic and electrostatic properties.

This thesis is about the design and implementation of a high performance Marching Cubes (MC) algorithm for computing and rendering molecular surfaces on a GPU. The main objective was to develop a high performance visualization program that allow us to know the most important aspects of molecular surfaces. This has involved a comparative study between a CPU-based implementation and a GPU-based implementation of the MC algorithm using a number of molecules retrived as ASCII files from PDB repository.

The main contributions of this thesis are the following:

- A multi-threaded GPU-based implementation of Marching Cubes algorithm to triangulate and rendering molecular surfaces.
- Computation of a van der Waals surface approximation from the Connolly surface by using triangle subdivisions together with a Newton corrector.

In a near future, we intend to work on a more scalable GPU-based algorithm that is capable of using a GPU cluster to triangulate and render very large molecules or even molecular complexes, with hundreds of thousands of atoms or above.

Bibliography

[1] Ballview. http://www.ballview.org/.

[2] Chemapplet. http://home.comcast.net/~g.purvis/gpurvis/ChemApp.html.

[3] Chemcraft. http://www.chemcraftprog.com/.

[4] Chimera. http://www.cgl.ucsf.edu/chimera/.

[5] Jmol. http://jmol.sourceforge.net/.

[6] NVIDIA CUDA Programming Guide 2.0. http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf.

[7] Rasmol. http://www.umass.edu/microbio/rasmol/index2.htm.

[8] Technical brief:NVIDIA Geforce GTX 200 GPU Architectural Overview. http://developer.download.nvidia.com.

[9] Viewmol. http://viewmol.sourceforge.net/.

[10] Vmd. http://www.ks.uiuc.edu/Research/vmd/.

[11] Daniele Agostino, Andrea Clematis, Ivan Merelli, Luciano Milanesi, and Matteo Coloberti. A grid service based parallel molecular surface reconstruction system. In *PDP '08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 455–462, Washington, DC, USA, 2008. IEEE Computer Society.

[12] Nataraj Akkiraju and Herbert Edelsbrunner. Triangulating the surface of a molecule. *Discrete Appl. Math.*, 71(1-3):5–22, 1996.

[13] Chandrajit Bajaj, Julio Candas, Vinay Siddavanahalli, and Zaiqing Xu. Compressed representations of macromolecular structures and properties. *Structure*, pages 463–471, 2005.

[14] Chandrajit Bajaj, Hangkyu Lee, R. Merkert, and Valerio Pascucci. Nurbs based b-rep models for macromolecules and their properties. In *SMA '97: Proceedings of the fourth ACM symposium on Solid modeling and applications*, pages 217–228, New York, NY, USA, 1997. ACM.

[15] Chandrajit Bajaj, Valerio Pascucci, Robert Holt, and Arun Netravali. Dynamic maintenance and visualization of molecular surfaces. In Mike Soss, editor, *Proceedings of the 10th Canadian Conference on Computational Geometry*, pages 68–69, Montréal, Québec, Canada, 1998. School of Computer Science, McGill University.

[16] James F. Blinn. A generalization of algebraic surface drawing. *ACM Trans. Graph.*, 1(3):235–256, July 1982.

[17] Jules Bloomenthal and Ken Shoemake. Convolution surfaces. *Computer Graphics*, 25(4):251–256, 1991.

[18] Frederic Cazals, Joachim Giesen, and Mark Pauly Afra Zomorodian. Conformal alpha shapes. *Eurographics Symposium on Point-Based Graphics*, 2005.

[19] Ho-Lun Cheng, Tamal K. Dey, Herbert Edelsbrunner, and John Sullivan. Dynamic skin triangulation. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 47–56, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[20] Ho-Lun Cheng and Xinwei Shi. Guaranteed quality triangulation of molecular skin surfaces. *IEEE Visualization*, October 2004.

[21] P. Cignoni, C. Montani, and R. Scopigno. Dewall: A fast divide and conquer delaunay triangulation algorithm in ed. 30(5):333–341, April 1998.

[22] Michael Connolly. Analytical molecular surface calculation. *Journal of Applied Crystallography*, 16(5):548–558, Oct 1983.

[23] Michael Connolly. Solvent-accessible surfaces of proteins and nucleic acids. *Science*, 221(4612):709–713, August 1983.

[24] Herbert Edelsbrunner. Deformable smooth surface design. *Discrete and Computational Geometry*, 21(1):87–115, January 1999.

[25] Herbert Edelsbrunner, M. Facello, Ping Fu, and Jie Liang. Measuring proteins and voids in proteins. *Hawaii International Conference on System Sciences*, 0:256, 1995.

[26] Herbert Edelsbrunner and Ernst Mucke. Three-dimensional alpha shapes. *ACM Trans. Graph.*, 13(1):43–72, 1994.

[27] R. Gabdoulline and R. Wade. Analytically defined surfaces to analyze molecular interaction properties. *Journal of Molecular Graphics*, 14:341–353, 1996.

[28] B. Gellatly and J. Finney. Calculation of protein volumes: an alternative to the voronoi procedure. *J. Mol. Biol.*, 161(2):305–322, October 1982.

[29] M. Gerstein, J. Tsai, and M. Levitt. The volume of atoms on the protein surface: calculated from simulation, using voronoi polyhedra. *J Mol Biol*, 249(5):955–966, June 1995.

[30] Jacob E. Goodman, Janos Pach, and Emo Welzl. *Combinatorial and Computational Geometry*. Cambridge University Press, 2007.

[31] J. Grant and B. Pickup. A gaussian description of molecular shape. *The Journal of Physical Chemistry*, 99(11):3503–3510, March 1995.

[32] Dan Halperin and Mark Overmars. Spheres, molecules, and hidden surface removal. In *SCG '94: Proceedings of the tenth annual symposium on Computational geometry*, pages 113–122, New York, NY, USA, 1994. ACM.

[33] Y. Harpaz, M. Gerstein, and C. Chothia. Volume changes on protein folding. *Structure*, 2(7):641–649, July 1994.

[34] Mark Harris, Shubhabrata Sengupta, and John Owens. Parallel prefix sum $scan$ with cuda. In GPU Gems 3, editor, *Hubert Nguyen*. Addison-Wesley Professional, Upper Saddle River, New Jersey, Aug 2007.

[35] M. Holst, N. Baker, and F. Wang. Adaptive multilevel finite element solution of the poisson-boltzmann equation i. algorithms and examples. *Journal of Computational Chemistry*, 21(15):1319–1342, 2000.

[36] B. Lee and F. Richards. The interpretation of protein structures: Estimation of static accessibility. *Journal of Molecular Biology*, 55(3):379–380, February 1971.

[37] M. Lee, M. Feig, F. Salsbury, and C. Brooks. New analytic approximation to the standard molecular volume definition and its application to generalized born calculations. *J Comput Chem*, 24(11):1348–1356, August 2003.

[38] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, volume 21, pages 163–169, New York, NY, USA, July 1987. ACM Press.

[39] N. Max. Approximating molecular surfaces by spherical harmonics. *Journal of Molecular Graphics*, 6(4):210, December 1998.

[40] B. McConkey, V. Sobolev, and M. Edelman. Quantification of protein surfaces, volumes and atom-atom contacts using a constrained voronoi procedure. *Bioinformatics*, 18:1365–1373, 2002.

[41] I. Merelli, L. Milanesi, D. Agostino, A. Clematis, M. Vanneschi, and M. Danelutto. Using parallel isosurface extraction in superficial molecular modeling. In *DFMA '05: Proceedings of the First International Conference on Distributed Frameworks for Multimedia Applications*, pages 288–294, Washington, DC, USA, 2005. IEEE Computer Society.

[42] Katalin Nadassy, Isabel Oliveira, Ian Alberts, Joel Janin, and Shoshana Wodak. Standard atomic volumes in double-stranded dna and packing of protein-dna interfaces. *Nuc. Ac. Res.*, 29:3362–3376, 2001.

[43] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[44] J. Pontius, J. Richelle, and S. Wodak. Deviations from standard atomic volumes as a quality measure for protein crystal structures. *J Mol Biol*, 264:121–136, 1996.

[45] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction (Monographs in Computer Science)*. Springer, August 1985.

[46] M. L. Quillin and B. W. Matthews. Accurate calculation of the density of proteins. *Acta Crystallogr D Biol Crystallogr*, 56(Part 7):791–794, July 2000.

[47] Frederic Richards. The interpretation of protein structures: Total volume, group volume distributions and packing density. *Journal of Molecular Biology*, 82(1):1–14, January 1974.

[48] Hanan samet. *Fondations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers, 2006.

[49] Roger Tam and Wolfgang Heidrich. Computing polygonal surfaces from unions of balls. *Proceedings of the Computer Graphics International*, 2004.

[50] M. Teschner and C. Henn. Mapping volumetric properties on molecular surfaces in real-time. In *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences*, page 265, Washington, DC, USA, 1995. IEEE Computer Society.

[51] M. Teschner, C. Henn, H. Vollhardt, S. Reiling, and J. Brickmann. Texture mapping: a new tool for molecular graphics. *J Mol Graph*, 12(2):98–105, 1994.

[52] Jerry Tsai and Mark Gerstein. Calculations of protein volumes: sensitivity analysis and parameter database. *Bioinformatics*, 18(7):985–995, 2002.

[53] H. Weiden, T. Goetze, and J. Brickmann. Fast generation of molecular surfaces from 3d data fields with an enhanced "marching cube" algorithm. *J. Comput. Chem.*, 14(2):246–250, 1993.

[54] J. Word, S. Lovell, J. Richardson, and D. Richardson. Asparagine and glutamine: using hydrogen atom contacts in the choice of side-chain amide orientation. *J. Mol. Biol.*, 285:1735–1747, 1999.

[55] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data Structure for Soft Objects. *The Visual Computer*, 2(4):227–234, February 1986.

[56] Wenqi Zhao, Guoliang Xu, and Chandrajit Bajaj. An algebraic spline model of molecular surfaces. In *SPM '07: Proceedings of the 2007 ACM symposium on Solid and physical modeling*, pages 297–302, New York, NY, USA, 2007. ACM.